Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Projektno vodenje razvoja programske opreme

Franc Solina

Ljubljana, 1997

CIP – Kataložni zapis o publikaciji
Narodna in univerzitetna knjižnica, Ljubljana

519.68:65.012(075.8)

SOLINA, Franc
Projektno vodenje razvoja programske opreme / Franc Solina. - 1. izd. - Ljubljana : Fakulteta za računalništvo in informatiko, 1997

ISBN 961-6209-12-4

71196672

Recenzent: prof. dr. Igor Kononenko
Lektor: Irena Roglič Kononenko, prof. slov.
Računalniško oblikovanje: Franc Solina

Založila: Fakulteta za računalništvo in informatiko, 1997
Urednik: mag. Peter Šega

Natisnil: FORMATISK Ljubljana
Naklada: 200 izvodov
1. izdaja

Po mnenju Ministrstva za šolstvo in šport Republike Slovenije št. 415-27/97 z dne 12. 12. 1997 gre za proizvod, od katerega se plačuje davek od prometa proizvodov po tarifni številki 3.

Kazalo

Slike

Tabele

Predgovor

Učbenik je namenjen študentom računalništva in informatike in drugim, ki se želijo seznaniti z osnovami programskega inženirstva. Programska oprema postaja v današnjem svetu nepogrešljiva za normalno delovanje človeške družbe. Zato je pravočasna izdelava pravilno delujoče nove, kakor tudi vzdrževanje obstoječe programske opreme, izrednega pomena.

V relativno kratkem obdobju razvoja računalništva je razvoj programske opreme le težko sledil silovitemu razvoju strojne opreme. Še posebej pa se niso dovolj hitro razvijale metode izdelave programske opreme, s katerimi bi lahko učinkovito kontrolirali njen razvojni cikel glede časa, kvalitete in stroškov. Ne le zaradi vloge, ki jo ima programska oprema v civilizirani družbi, marveč predvsem zato, ker je razvoj programske opreme danes ena od najhitreje razvijajočih se in dobičkanosnih gospodarskih dejavnosti, se metode razvoja programske opreme izredno hitro izpopolnjujejo in spreminjajo. Prehod od individualnega oziroma obrtnega k industrijskemu načinu izdelovanja, ki je na drugih področjih zahteval cele generacije, na področju izdelave programske opreme poteka le nekaj deset let, pa tudi povsem končan še ni. V primerjavi z drugimi industrijskimi panogami je izdelavo programske opreme težje načrtovati, težje pa je tudi kontrolirati kvaliteto njenih izdelkov. Programska oprema ima pač svoje posebnosti in zato se je razvila nova disciplina programskega inženirstva, ki si prizadeva razvoj programske opreme postaviti ob bok drugim inženirskim disciplinam.

V tem učbeniku so podane osnove programskega inženirstva. Zaradi hitrega razvoja je stanje na tem področju precej heterogeno. Razvijalci lahko izbirajo različne pristope in metode izdelave programske opreme glede na zahteve uporabnikov pa tudi glede na svoje izkušnje. Zato bo bralec te knjige zaman iskal absolutno veljavne rešitve. Do kvalitetne programske opreme vodijo različne poti. Najnovejše metode niso vedno najprimernejše v danih okoliščinah. Zato je za programskega inženirja pomembno, da pozna vsaj osnove čimveč različnih metod in principov, ki jih lahko po potrebi med seboj tudi kombinira. Iz poznavanja različnih metod in tega, kako

so se razvijale, pa lahko tudi lažje kritično ovrednotimo vse nove metode, ki se pojavljajo na tem področju. V učbeniku so zato podane osnove tako zdaj že klasičnih metod kot tudi novejše objektno usmerjene metode razvoja programske opreme. Za praktično delo po izbrani metodologiji bo vsakdo moral poseči še po dodatni literaturi, predvsem pa bo moral zbrati čimveč izkušenj pri samem delu, po možnosti v skupini, kjer mu bodo za zgled bolj izkušeni sodelavci.

Knjiga je deloma nastala po predlogah za predavanja in mestoma po knjigi "Organizacijski, psihološki in sociološki vidiki projektnega dela", ki sem jo pred leti napisal z dr. Francem Križajem. Razdeljena je na 12 poglavij. Uvodu, v katerem je predstavljena vloga programske opreme v današnjem svetu, sledi poglavje, v katerem so podane nekatere osnovne lastnosti programskega inženirstva. V tretjem poglavju je predstavljen projektni način dela, ki ga razvoj programske opreme zahteva, v četrtem psihološki in sociološki vidiki projektnega dela, v petem pa za projektni način dela izjemno pomembno mrežno načrtovanje. V šestem poglavju so predstavljene posebnosti projektov za razvoj programske opreme, kot na primer, kako se ocenjuje potrebno delo pri teh projektih in kako se skrbi za kvaliteto. V nadaljnjih petih poglavjih knjiga sledi klasičnemu razvojnemu ciklu programske opreme, v katerem si sledijo Analiza zahtev, Načrtovanje, Kodiranje, Testiranje in Vzdrževanje, saj vse druge metode razvoja le po drugem vrstnem redu ali na drug način kombinirajo iste aktivnosti.

V Ljubljani, november 1997 Franc Solina

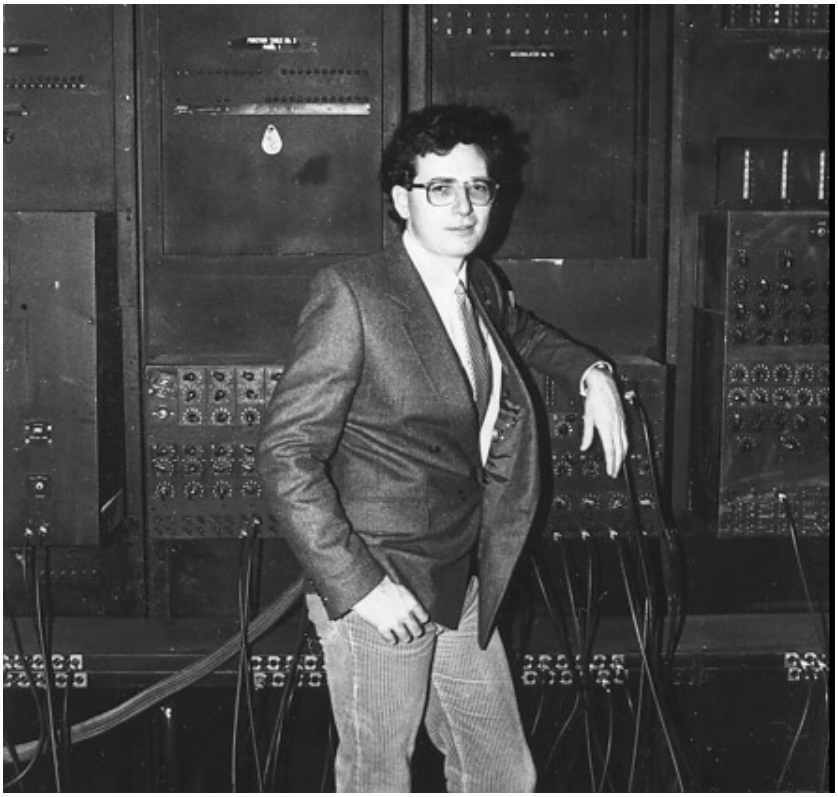
Poglavje 1

Uvod

Vloga programske opreme v današnjem svetu postaja vse večja [15]. V pionirski dobi računalništva je bilo osnovno gibalo razvoj računalniške strojne opreme. Programska oprema je bila pač nadgradnja, ki se je povsem podrejala strojni opremi. Programiranje pa se je obravnavalo bolj kot *umetnost* in ne kot inženirsko delo [37]. Cena procesiranja in shranjevanja podatkov je bila glavna ovira širši uporabi računalnikov. Z ogromnim napredkom mikroelektronike v zadnjih dveh desetletjih je postala strojna oprema veliko zmogljivejša, predvsem pa veliko cenejša. Vloga programske opreme se je v tej novo nastali situaciji povsem spremenila, saj je šele z ustrezno programsko opremo možno izkoristiti vse zmožnosti strojne opreme. Zato je danes osrednji cilj večine računalniške industrije zmanjšati ceno razvoja programske opreme in izboljšati njeno kvaliteto. Toda razvoj ustrezne programske opreme je v celotnem svetu v krizi [2, 51]. Primanjkuje ustreznih kadrov, celo vodilna podjetja za razvoj programske opreme kasnijo z dobavo novih produktov ali z novimi verzijami obstoječe programske opreme. Tudi po dobavi kupcem se še pojavljajo resne napake. Kljub velikim naporom, se te slabosti pri razvoju programske opreme še prisotne, saj klasična pravila upravljanja na tem področju ne veljajo. Proces razvijanja programske opreme je težko obvladovati in nadzorovati.

1.1 Spreminjajoča se vloga programske opreme

V pionirski dobi računalnikov je bila programska oprema povsem unikatna, saj je bil vsak računalnik unikaten izdelek. Vsak računalnik se je programiral na svojski način. Tako so ENIAC, prvi pravi elektronski računalnik, ki so ga zgradili na University of Pennsylvania v Philadelphiji, programirali s

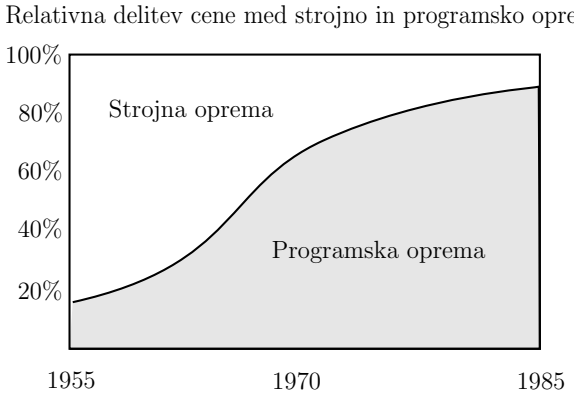


Slika 1.1: Avtor knjige pred računalnikom ENIAC med svojim doktorskim študijem na University of Pennsylvania leta 1984. V ozadju so dobra vidna stikala in kabli, s katerimi se je programiral ENIAC.

premikanjem stikal in pretikanjem konektorjev (slika 1.1). Šele konec 50-tih let se v novi (drugi) generaciji računalnikov, zgrajenih na tranzistorski tehnologiji, pojavi več primerkov iste vrste računalnika in s tem tudi standardizirana programska oprema. Kljub temu pa igra programska oprema v tedanjih računalniških rešitvah še vedno drugorazredno vlogo. Zaradi visoke cene strojne opreme je bilo potrebno programsko opremo prilagajati tako, da je bila strojna oprema čimbolj izkoriščena. S tretjo generacijo računalnikov v 60-tih letih, ki je že temeljila na integriranih vezjih, se je število računalnikov skokovito povečalo. Prvi računalniki so se pojavili tudi v Sloveniji. Računalniki so se manjšali in se počasi začeli seliti iz posebnih računalniških centrov k njihovim dejanskim uporabnikom. Pojavili so se novi koncepti v programski opremi, kot sta večopravilnost in večuporabnost

ter številni novi programski jeziki. Nadaljnjo veliko povečanje gostote integriranih vezij (LSI, VLSI) in zmanjšanje stroškov izdelave vodi do četrte generacije računalnikov v 70-tih letih. Mikroprocesorji v 80-tih letih omogočijo nastanek osebnih računalnikov, kar število računalnikov skokovito poveča. S tem se je tržišče za programsko opremo tudi izredno povečalo in spremenilo. Nastanejo nova podjetja, ki se usmerijo izključno v razvoj programske opreme in to opremo tržijo kot masovni produkt. Vedno večjo vlogo igra enostavnost uporabe programske opreme, zato grafični uporabniški vmesniki postanejo vsakdanji.

Kako se je spreminjala vloga programske opreme, nazorno kaže razmerje med ceno strojne in programske opreme za nek po naročilu razvit sistem na sliki 1.2.



Slika 1.2: Delež programske opreme v ceni po naročilu razvitih informacijskih sistemov vztrajno raste.

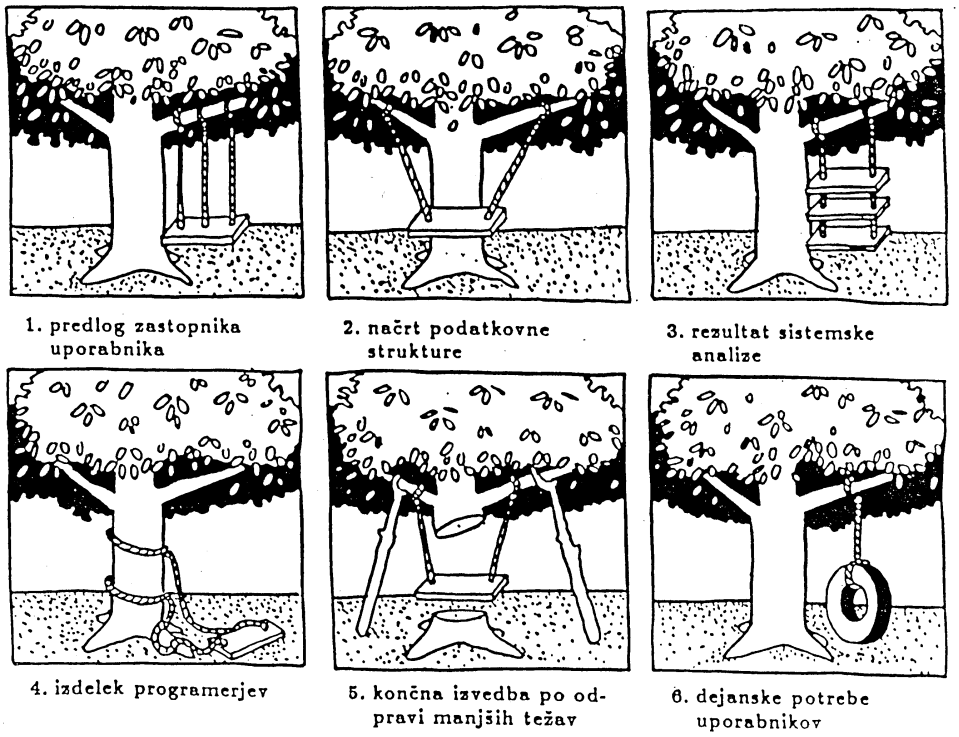
Zadnja velika in morda od pojava računalnikov celo največja prelomnica v razvoju računalništva pa je pojav globalnega računalniškega omrežja, to je interneta ali medmrežja. Povezava računalnikov v to medmrežje ima poleg velikih informacijskih, ekonomskih in družbenih posledic tudi velik vpliv na programsko opremo. Pojavljajo se nove vrste programske opreme, novi programski koncepti, novi načini uporabe, testiranja in distribucije programske opreme.

1.2 Znaki krize programske opreme

Računalniki so danes prisotni v skoraj vseh dejavnostih civilizirane človeške družbe. Če odpovedo računalniki, ne moremo do svojega denarja v banki, ne morejo nam izdati osebnih dokumentov ali nas vpisati v šolo, lahko zastane promet in še veliko drugih človeških aktivnosti. Računalniki se danes ne uporabljajo le tam, kjer so nadomestili zamudno ročno delo ali kjer so izboljšali že prej obstoječe dejavnosti. Računalniki so omogočili storitve in rešitve, ki si jih brez računalnikov sploh ne bi mogli zamisliti, in spodbudili celo vrsto novih industrijskih panog in storitvenih dejavnosti. Celotna človeška družba je zato zelo odvisna od pravilnega delovanja računalniških sistemov. Da bi ti sistemi pravilno delovali, pa se morajo sproti prilagajati spremembam v svojem okolju, saj se človeška družba neprestano razvija. Razvijati je potrebno novo programsko opremo za nove dejavnosti in potrebe, obstoječo programsko opremo pa prilagajati spremembam in jo občasno tudi nadomestiti z novo. Zato potrebe po programski opremi skokovito naraščajo. Razvoj programske opreme pa ni sposoben slediti zahtevam tržišča. Zaradi slabe kvalitete in nezadostnih zmogljivosti tudi ni možno dovolj dobro vzdrževati obstoječe programske opreme. Storilnost programerjev ne narašča dovolj hitro, zato jih stalno primanjkuje. S poslovnega vidika pa je osnovni problem to, da je veliko težje kot v drugih dejavnostih vnaprej ocenjevati potreben čas in zmogljivosti za razvoj nove programske opreme.

Zakaj je prišlo do takega položaja? Poleg že omenjenih vedno večjih potreb po programski opremi se sama tehnologija razvoja nenehno spreminja. Preden razvijalci obvladajo neko novo metodologijo, se že pojavi nova in boljša. Zato se programerji običajno nikoli ne naučijo vseh podrobnosti neke metode ali orodja za razvoj programske opreme. Ker so v stalni časovni stiski, pogosto tudi ne opravijo dosledno analize in načrtovanja, temveč prehitro začnejo kar s programiranjem.

Tudi organizacijski vidiki razvoja so težko obvladljivi. Nove metodologije razvoja običajno zahtevajo tudi drugačno organizacijo in vodenje razvoja. Zaradi relativne mladosti programskega inženirstva je malo izkušenj, pa še te zaradi uvajanja novosti hitro zastarajo. V veliko pomoč tudi niso razni standardi in formalne metode, ki se jih sicer pojavlja vedno več. Tisti, ki vodijo razvojne skupine pogosto tudi dajejo večji pomen najnovejši strojni opremi, ne pa raznim razvojnim programskim orodjem, za katere je potrebno ljudi še dodatno izšolati. Najpogostejša napaka pri vodenju in upravljanju razvojnega projekta, ki kasni, je dodajanje novih programerjev. To ima običajno ravno nasproten učinek od želenega, saj namesto da bi opravljali



Slika 1.3: Znaki krize v razvijanju programske opreme se odražajo celo v karikaturah.

svoje delo, stari člani skupine uvajajo v projekt nove člane in tako delo še bolj zamuja [9]. Pogosto je za težave pri večjih projektih kriva tudi slaba komunikacija med programerji in vodstvom, ki zato ne pozna dejanskega stanja programske kode. Koliko programov, ki so po zagotovilih programerjev skoraj že delali pravilno, ni bilo pravočasno ali celo nikoli končanih?

Uporabniki oziroma naročniki programske opreme se pogosto tudi ne zavedajo posebnosti v razvoju programske opreme. Ne znajo jasno izraziti svojih zahtev, radi zahtevajo spremembe ali dopolnitve pozno med razvojem, saj si predstavljajo, da je programska oprema, ker pač ni fizičen izdelek, poljubno fleksibilna. V resnici so cene sprememb med razvojem dvakrat do šestkrat dražje, med vzdrževanjem pa celo do stokrat dražje kot med definiranjem zahtev na začetku razvoja.

Vse te težave odražajo postopen in še vedno potekajoč prehod razvoja programske opreme od neke vrste umetnosti in kasneje obrti k industrijski

proizvodnji. Pri tem prehodu se žal ni možno neposredno zgledovati pri drugih dejavnostih, saj je programska oprema zelo specifičen izdelek.

1.3 Značilnosti programske opreme kot izdelka

V ceni industrijskega produkta, ko je na primer avtomobil, je običajno le majhen del namenjen za poplačilo razvojnih stroškov tega modela, največji delež je za njegovo dejansko izdelavo. Večina stroškov v zvezi s programsko opremo pa nastane med njenim razvojem, saj kasnejšo produkcijo predstavlja le kopiranje na ustrezni medij. Programska oprema se tudi ne izrabi tako kot fizični izdelki, vendar vseeno zastari. Zato je potrebno programsko opremo vzdrževati, kar pa je bolj zahtevno kot pri strojni opremi, saj za programsko opremo nimamo "rezervnih delov". Večina programske opreme ni sestavljena iz že obstoječih elementov, ampak je zgrajena "po naročilu". Morda največja posebnost programske opreme pa je to, da imajo lahko že *zelo majhne spremembe zelo velike posledice*. V drugih industrijskih produktih so se naučili narediti kritične dele dovolj robustno in skušajo produkt tudi sestavili tako, da kljub okvaram nekaterih sestavnih delov celota ohrani večino svoje funkcionalnosti. V programski opremi pa tega žal še ne znamo doseči.

Kot primer take občutljivosti je poučen naslednji primer [4]. V Fortranskem programu, ki je kontroliral vesoljsko sondo Mariner na poti k Veneri, je zgolj zamenjava vejice s piko povzročila njeno izgubo. Namesto `DO 3 I = 1,3` je v programu pisalo `DO 3 I = 1.3` in prevajalnik je namesto zanke, ki naj bi se trikrat izvedla, pripisal spremenljivki `DO3I` vrednost `1.3`.

Če upoštevamo, da veliki sistemi obsegajo tudi do milijon vrstic kode, si ni težko predstavljati, da tudi po obsežnem testiranju v kodi še ostanejo napake. Nenazadnje lahko vsakdo prebere opozorila, običajno natisnjena v drobnem tisku, ki so priložena vsakemu programskemu paketu za široko potrošnjo, v katerem proizvajalci izrecno opozarjajo kupca, da ne jamčijo niti za uporabnost niti za neposredno ali posredno škodo, ki bi jo povzročila uporaba kupljene programske opreme[1].

[1] We make no warranty or representation, either express or implied, with respect to the software described in this manual, its quality, performance, merchantability, or fitness for any particular purpose. As a result, the software is sold 'as is', and you the purchaser are assuming the entire risk as to its quality and performance. In no event will we be liable for direct, indirect, special, incidental, or consequential damages resulting from any defect in the software or manual, even if we have been advised of the possibility of such damages.

Poglavje 2

Programsko inženirstvo

Kriza programske opreme, ki smo jo opisali v uvodnem poglavju, je postala pereča že v 60-tih letih. Leta 1968 so na konferenci pod pokroviteljstvom organizacije NATO prvič skovali izraz "Software Engineering" ali po slovensko programsko inženirstvo [43]. Izraz je bil prvotno mišljen nekoliko provokativno, da bi odgovorili na dilemo, ali je možno programsko opremo zgraditi na podoben način, kot se gradijo mostovi in hiše, s pomočjo zdrave teoretične podlage in preizkušenih konstrukcijskih metod, tako kot v drugih inženirskih disciplinah[1].

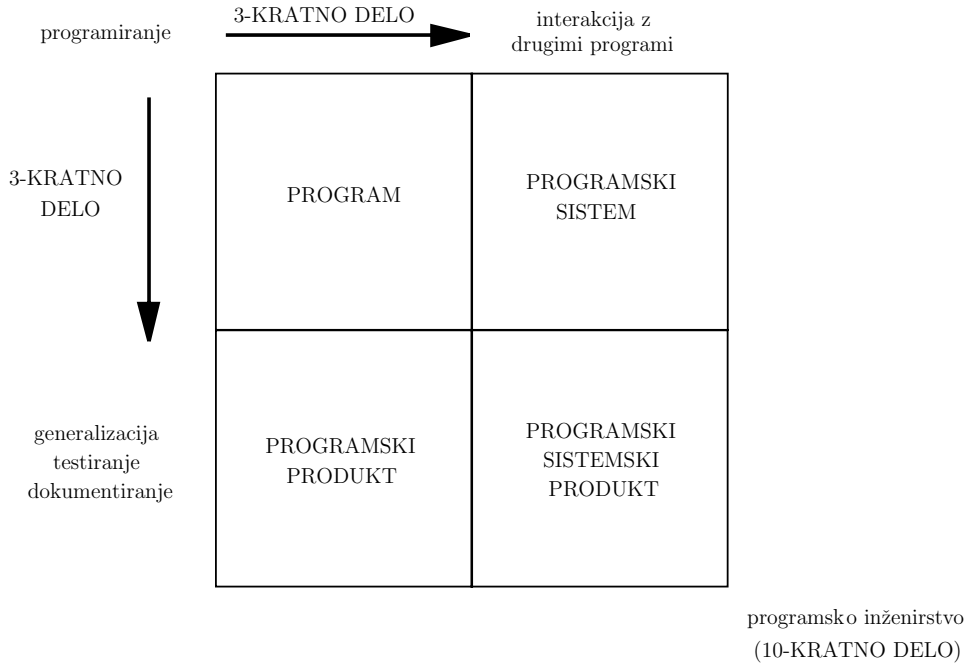
2.1 Kaj je cilj programskega inženirstva?

Definicije programskega inženirstva so številne, toda njegove bistvene značilnosti so naslednje:

1. Osnovni cilj programskega inženirstva je *uporabna in kvalitetna* programska oprema. Programska oprema, ki je rezultat sistematičnega razvojnega dela, je veliko več kot računalniški program, ki je le rezultat programiranja, kar je prikazano tudi na sliki 2.1.

   To, kar si postavljamo za cilj programskega inženirstva — *programski sistemski produkt*, zahteva tipično desetkrat več dela kot program, napisan za enkratno uporabo. Sistemski programski produkt naj bi bil ustrezno testiran, dokumentiran in povezljiv v širšo celoto. Poleg

[1] Software engineering is the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines [43].



Slika 2.1: Razlika med programiranjem in programskim inženirstvom je očitna.

tega naj bi ga bilo tudi enostavno vzdrževati in uporabljati. Vse to so vidiki kvalitete, o katerih bo govora še kasneje v knjigi.

2. Programsko inženirstvo zanima predvsem *gradnja velikih sistemov.* Ločimo namreč lahko med programiranjem na malo in na veliko. Kje je meja, je težko točno definirati. Program, dolg 100 vrstic, je očitno majhen, 50.000 vrstic dolg program pa je velik program. O programiranju na malo govorimo pri programih, ki jih napiše posameznik v nekaj dneh. Tradicionalne tehnike in orodja za programiranje so namenjena prav takemu načinu programiranja. Te tradicionalne metode pa je težko direktno nadgraditi za razvoj veliki sistemov, kjer ne gre le za veliko večje programe, ampak običajno tudi za večje število med seboj povezanih programov.

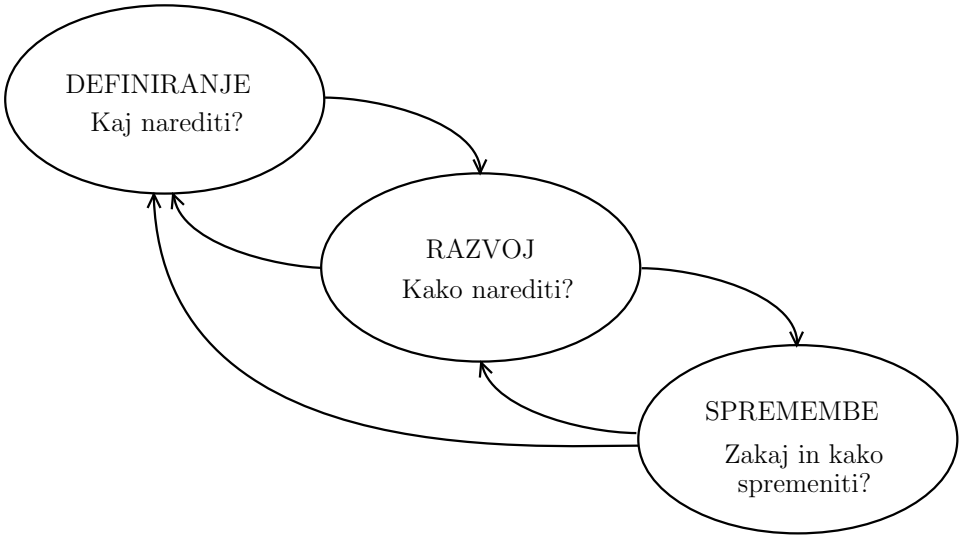
3. Pri velikih sistemih je težko hkrati nadzorovati vse podrobnosti. *Kompleksnosti* se običajno lotimo tako, da večje probleme razdelimo na manjše in obvladljive. Delitev na podprobleme pa mora biti taka, da so ločnice med podproblemi čimbolj jasne in preproste. V pro-

gramskem inženirstvu tako delitev dosežemo s programskimi moduli, ki skrbijo za določeno nalogo. Velika večina programskih sistemov ni kompleksna zaradi kompleksnosti problema, ki ga rešuje, temveč zato, ker mora hkrati obvladovati veliko število podrobnosti.

4. Pri razvoju velikih sistemov, ki zahtevajo veliko časa in ljudi, je *učinkovitost* izrednega pomena. Potrebno je kontrolirati stroške in čas razvoja, saj skupine za razvoj programske opreme tekmujejo pri tem z drugimi razvojnimi skupinami. Potrebe po novih aplikacijah pa tudi prekašajo obstoječe zmogljivosti.

5. Za razvoj programske opreme se je zato uveljavil *projektni način dela.* Projektni način dela omogoča skrbno načrtovanje dela in njegovo delitev med posameznimi razvijalci, kontrolo časa, stroškov in drugih zmogljivosti. Omogoča tudi delitev odgovornosti, organizacijo komunikacij in nadzor kvalitete opravljenega dela. Zato so naslednja tri poglavja te knjige namenjena projektnemu načinu dela in mrežnemu načrtovanju kot enemu od bistvenih elementov razvoja programske opreme.

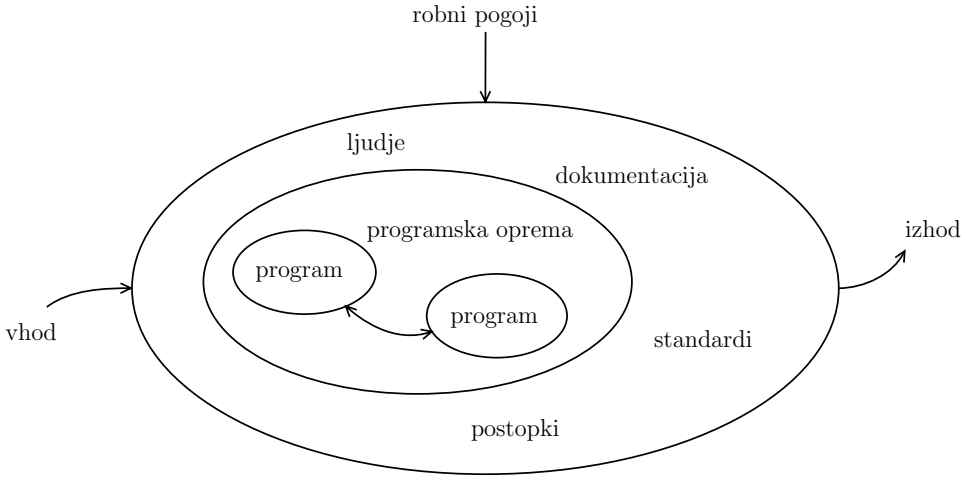
2.2 Razvojni cikel



Slika 2.2: Tri generične faze razvoja programske opreme

Razvoj programske opreme poteka v treh generičnih fazah: *definiranje,*

razvoj in spremembe (slika 2.2). V fazi definicije je potrebno predvsem čimbolj točno ugotoviti, kaj naj bi bile naloge predvidene programske opreme in kako se naj programska oprema vključi v širše okolje. V razvojni fazi gre za izdelavo produkta, ki naj zadosti definiranim zahtevam. S to fazo se pravzaprav projekt razvoja programske opreme v ožjem smislu tudi konča, saj tretja faza, to je vzdrževanje programske opreme, traja ves čas uporabe programske opreme, to pa je lahko tudi več kot deset let. V tem času je povsem normalno, da se porodijo zahteve po spremembah. Bolj obširne spremembe programske opreme nato lahko obravnavamo kot nove projektne naloge.



Slika 2.3: Informacijsko načrtovanje določa, kateri del nekega problemskega okolja naj podpre programska oprema in kako se bo ta oprema povezovala z drugimi deli tega okolja.

V vsaki generični fazi je združenih več aktivnosti. V fazi definiranja gre predvsem za to, da se ugotovi, kaj naj bo cilj projekta. Izdelati je potrebno tudi projektni načrt, po katerem bodo potekale vse nadaljnje aktivnosti. Projektni načrt lahko po potrebi kasneje dopolnjujemo, ko se izluščijo nadaljnje podrobnosti. Kot prva aktivnost v fazi definiranja je lahko vključen tudi sistemski inženiring ali informacijsko načrtovanje (slika 2.3). Sistemski inženiring je neke vrste meta-projekt, ki naj odgovori na vprašanje, katere probleme naj sploh vključimo v projekt in kako se ti problemi navezujejo na svojo širšo okolico. Tu gre za to, da ugotovimo pravo kombinacijo strojne in programske opreme in njuno povezavo oziroma vmesnike z obstoječimi elementi okolja v katerem bo programska oprema delovala (opre-

ma, ljudje, podatkovne baze, postopki, standardi itd.).

V razvojni fazi poteka izdelava programske opreme, ki zahteva njeno načrtovanje, kodiranje in testiranje. V fazi sprememb pa gre za različne vrste vzdrževanja, od popravljanja napak do prilagajanja programske opreme novim zahtevam in okoliščinam.

V praksi in teoriji se je uveljavilo kar nekaj postopkov razvoja programske opreme [51, 65], ki bolj natanko določajo, kako se naj posamezne razvojne aktivnosti vrstijo in povezujejo med seboj. Vsi ti postopki ali metode razvoja imajo svoje prednosti pa tudi pomanjkljivosti. Pogosto jih tudi kombiniramo, da bi čimbolj izkoristili njihove prednosti v dani situaciji.

2.2.1 Aktivnosti v razvoju programske opreme

V različnih postopkih razvoja programske opreme gre za različne vrste kombiniranja predvsem naslednjih aktivnosti:

- **Analiza zahtev programske opreme.** Cilj analize je popolno razumevanje nalog in lastnosti programske opreme. Vse zahteve je potrebno skrbno dokumentirati in se o njih pogovoriti z naročnikom.

- **Načrtovanje** programske opreme je večstopenjski proces, ki zahteve s pomočjo uporabe podatkovnih in kontrolnih struktur ter programske arhitekture prevede v načrt programske opreme. Načrt je taka predstavitev programske opreme, da njene lastnosti lahko ocenimo še preden se začne kodiranje.

- **Kodiranje** je ob popolnem načrtu zgolj mehanske narave, saj naj bi že načrtovanje definiralo vse potrebne podrobnosti.

- **Testiranje** ni le izolirana aktivnost, ki preverja rezultate kodiranja, temveč se mora izvajati ves čas razvoja programske opreme. Rezultate vseh aktivnosti je potrebno preveriti, saj prej ko odkrijemo neko napako ali pomanjkljivost med razvojem, lažje in ceneje jo lahko odpravimo. Verifikacija je preverjanje pravilnosti rezultatov posamezne faze v razvoju. Validacija pa vzame v obzir širši vidik in preverja, če so rezultati posamezne faze tudi skladni z osnovnimi zahtevami, postavljenimi na začetku razvoja.

  Ko testiramo programsko kodo, je vsak element ali modul programske kode potrebno testirati posebej, med integracijo pa postopno, po vsakem dodanem modulu. Po končani integraciji sledi še cela vrsta sistemskih testiranj.

- **Vzdrževanje** programske opreme je potrebno zaradi odpravljanja napak po končanem razvoju in zaradi spremenjenih zunanjih okoliščin, v katerih sistem deluje.

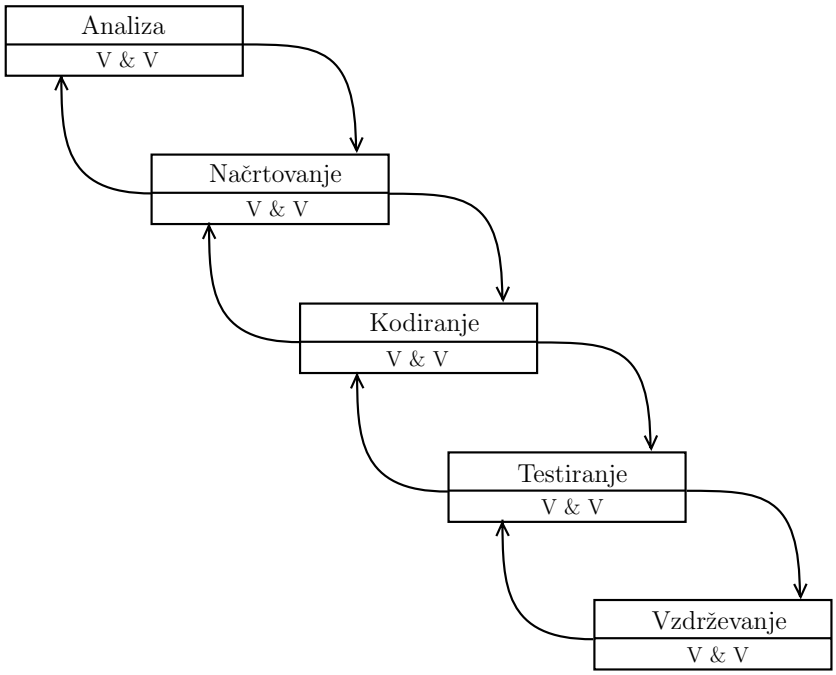
2.3 Razvojni modeli programske opreme

2.3.1 Klasični razvojni cikel

Klasični, linearni ali kaskadni življenjski cikel je najstarejši in še vedno pogost postopek razvoja programske opreme. Zgleduje se po standardnem postopku razvoja tehničnih izdelkov. Vse aktivnosti si sledijo zaporedno (slika 2.4). Ker se v klasičnem razvojnem modelu ni možno vračati na predhodne faze, je nujno, da so zahteve že na samem začetku popolno in nedvoumno definirane. To zahteva skrbno analizo in sodelovanje naročnika ali uporabnika. Da ne bi med razvojem prišlo do prevelikega odstopanja med sistemom, ki ga razvijamo in zahtevami uporabnikov, je smiselna po vsaki fazi poleg verifikacije rezultatov tudi njihova validacija. Edino tak način razvoja je bil v preteklosti ekonomičen, saj so bile vse aktivnosti od načrtovanja do kodiranja in testiranja zaradi še neobstoječih orodij za podporo razvoja programske opreme zelo zamudne.

V praksi pa imajo projekti redkokdaj popolnoma sekvenčno naravo. Posebej individualni razvijalci radi med razvojem skačejo med različnimi ravnmi abstrakcije, od sistemskih funkcij do podrobnosti v programski kodi. Posamezne aktivnosti, ki naj bi bile med seboj strogo ločene in zaporedne, se tako med seboj prekrivajo in mešajo. Za to so krive tudi nejasne ali nepopolne zahteve na začetku projekta. To je lahko zaradi same narave projekta ali zato, ker naročnik nima pravega vpogleda v naravo razvoja programske opreme. Pri klasičnem razvojnem modelu mora biti naročnik tudi potrpežljiv, saj je rezultat dela razviden šele na samem koncu razvojnega cikla. Bistvene pomanjkljivosti se tako lahko pokažejo šele na koncu razvoja, ko postane vsaka sprememba izredno draga.

Kljub naštetim pomanjkljivostim je klasični razvojni cikel primeren za rutinske projekte, še posebej takrat, ko ima razvojna skupina že dovolj izkušenj na določenem aplikacijskem področju. Zaradi pojava novih razvojnih orodij, ki omogočajo bolj eksperimentalen način dela, pa so se pojavili novi razvojni postopki.

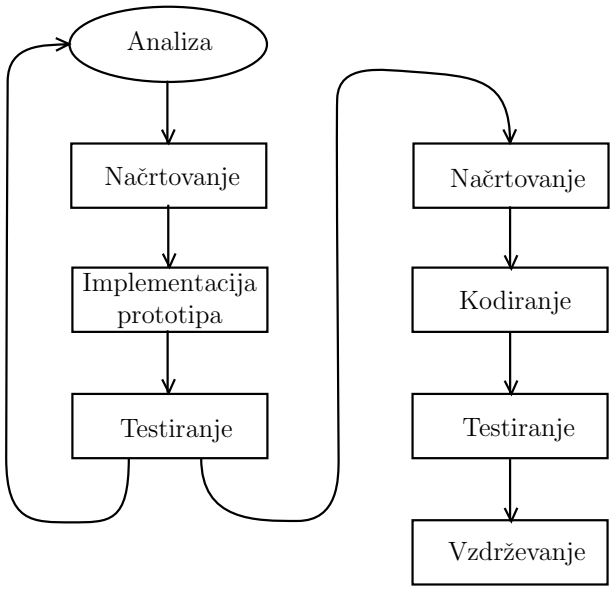


Slika 2.4: Klasični razvojni cikel programske opreme. V & V: verifikacija in validacija. Verifikacija pomeni preverjanje, če gradimo sistem pravilno — predvsem, če so pravilni prehodi med posameznimi fazami razvoja. Z validacijo ugotavljamo, ali gradimo pravi sistem — sistem, ki ustreza potrebam.

2.3.2 Razvoj z uporabo prototipov

Osnovni problem klasičnega razvojnega cikla je, da je na samem začetku razvoja težko nedvoumno in popolno definirati zahteve. Če vzamemo v obzir le obstoječo situacijo kot izhodišče za analizo, zanemarimo možnosti izboljšav, ki jih nudi avtomatizacija. Kadar gre za povsem novo aplikacijo, je poleg formulacije zahtev težko točno določiti, kaj so vhodni podatki, ne ve se, kako zanesljivi so algoritmi, niti kakšen naj bi bil uporabniški vmesnik med načrtovanim sistemom in uporabniki. V takih primerih je smiselno uporabiti razvojni postopek s pomočjo uporabe prototipov (slika 2.5), kar pomeni, da najprej zgradimo enega ali več *prototipov* ali *modelov programske opreme,* ki naj razjasnijo, kakšno programsko opremo pravzaprav potrebujemo.

Razvoj s pomočjo prototipov je sicer običajen pri razvoju novih produktov. Za nove fizične produkte, kot so avtomobili ali mikroprocesorski čipi se izdela cela vrsta prototipov, preden se začne z redno proizvodnjo. V ceni



Slika 2.5: Razvoj programske opreme z uporabo prototipov

fizičnih izdelkov pa je zajeta predvsem cena izdelave posameznih fizičnih kopij. Pri programskih produktih pa je kopiranje, to je izdelovanje kopij, praktično zastonj. Skoraj vsi stroški programskega produkta nastanejo med samim razvojem produkta. Zato ne bi bilo smiselno, da bi naredili več povsem funkcionalnih prototipov, ki bi jih nato zavrgli. Programski prototip mora biti karseda hitro in poceni narejen. To lahko dosežemo na več načinov:

- S pomočjo raznih razvojnih orodij (npr. jezikov 4. generacije) hitro generiramo kodo, ki sicer ni učinkovita, vendar omogoča izvajanje in testiranje.
- Izdelamo prototip, v katerega implementiramo le nekatere funkcije, predvsem tiste, ki jih želimo bolj natančno definirati. Druge funkcije pa izpustimo ali le simuliramo.
- Uporabimo podoben, že obstoječ produkt, s pomočjo katerega lažje opišemo funkcije in opozorimo na razlike, ki jih pričakujemo v novem produktu.

Uporaba prototipov je posebej primerna takrat, ko zahteve niso povsem jasno določene. S pomočjo prototipa lahko te nejasnosti odpravimo še preden

investiramo veliko dela v izdelavo pravega produkta. Pogosto se prav s pomočjo prototipa definira uporabniški vmesnik.

V postopku razvoja s pomočjo prototipov ločimo dve osnovni fazi, izdelavo in ocenjevanje prototipa, ki se lahko večkrat ponovita, in dejansko izdelavo programskega produkta. Analiza poteka kot običajno tako, da posebej identificiramo tiste elemente, ki še niso povsem dorečeni. Načrtovanje prototipa se nato osredotoči prav na te elemente. Prototip moramo testirati in oceniti s pomočjo uporabnika oziroma naročnika. Če je potrebno, se ves proces izdelave prototipa in ocenjevanja ponovi večkrat. Zaradi prihranka časa se naj pri gradnji prototipa uporabi čimveč orodij za hitro generacijo kode, saj učinkovitost in kvaliteta prototipa nista bistvena. To postane važno šele pri izdelavi končnega produkta, ki lahko poteka tudi na klasičen razvojni način. V tem primeru je prototip predvsem orodje pri analizi zahtev.

Obstaja pa še drug pristop k razvoju s pomočjo prototipov, pri katerem se prototip v nekaj iteracijah postopno razvije v končni produkt. Toda izkušnje kažejo, da takšni produkti niso robustni, dokumentacija je ponavadi pomanjkljiva, zaradi slabše kvalitete tako izdelane programske opreme je tudi kasnejše vzdrževanje težje. Zaradi hitrega razvoja posameznega prototipa ni možno skrbno sproti izvajati vseh aktivnosti, kasneje pa je zelo težko dodati kvaliteto končnemu produktu.

Pri razvoju s pomočjo prototipov moramo zato upoštevati naslednje izkušnje:

1. Naročnik in razvijalec se morata vnaprej dogovoriti o vlogi prototipa v postopku razvoja. Ta je največkrat le **mehanizem za specifikacijo zahtev**, ki ga nato zavržemo, končni produkt pa dosledno in skrbno izdelamo na novo. Ko naročnik namreč vidi delujoči prototip programske opreme, ga ima lahko že za skoraj gotov izdelek, ki morda potrebuje le še nekaj dodelav. Ne razume, da je prototip v tem primeru le na hitro zgrajena hišica iz kart.

2. Kompromisi, ki smo jih zaradi kratkega časa in poceni izvedbe povsem upravičeno sprejeli pri gradnji prototipa (npr. izbira programskega jezika), se po inerciji ne smejo prikrasti v končni produkt in postati stalni del sistema.

3. Izdelavo in ocenjevanje prototipov je potrebno načrtovati in kontrolirati. Število prototipov pa je smiselno že vnaprej omejiti.

2.3.3 Razvoj s 4. generacijo programskih jezikov

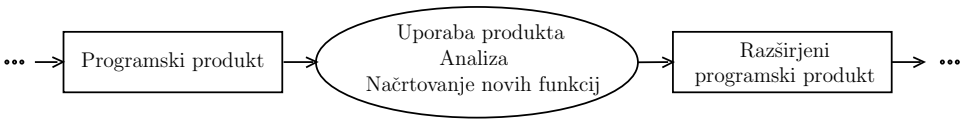
Z imenom *4. generacija programskih jezikov* označujemo številna orodja za hitro specifikacijo programskih produktov. Ta orodja so znana tudi kot aplikacijski generatorji (angl. application generators). Bistvo teh orodij je specifikacija *nekaterih* lastnosti na zelo *visokem* nivoju, orodja pa potem na osnovi te specifikacije avtomatično generirajo kodo. Prednost specifikacije na visokem nivoju je predvsem veliko hitrejši razvoj. Žal pa so ta orodja prav zaradi visokega nivoja specifikacije tudi omejena na *zelo specifična* področja. 4. generacija programskih jezikov so neproceduralni jeziki, ki so najpogosteje namenjeni bodisi za zbiranje podatkov iz podatkovnih baz bodisi za generiranje poročil in manipuliranje s podatki.

Poleg ozkega področja uporabe predvsem za poslovne aplikacije ima razvoj programske opreme s 4. generacijo programskih jezikov še naslednje pomanjkljivosti [13]:

- S 4. generacijo programskih jezikov ni bistveno lažje programirati kot z višjimi programskimi jeziki.
- Čas za analizo in načrtovanje tudi ni bistveno krajši kot pri razvoju z višjimi programskimi jeziki.
- Koda je počasna v primerjavi s proceduralnimi programskimi jeziki.
- Kadar se kakšne funkcije ne da izraziti jeziku 4. generacije, lahko sicer funkcijo napišemo v jeziku 3. generacije. Toda vzdrževanje take mešano nastale kode je zelo težavno.

Kljub temu pa programski jeziki 4. generacije postajajo vedno boljši in univerzalnejši. Njihova uporaba se je predvsem na področju poslovnih in informacijskih sistemov precej razmahnila, smiselna pa je tudi za hitro izdelavo prototipov v okviru razvoja programske opreme s pomočjo prototipov.

2.3.4 Postopni razvoj programske opreme



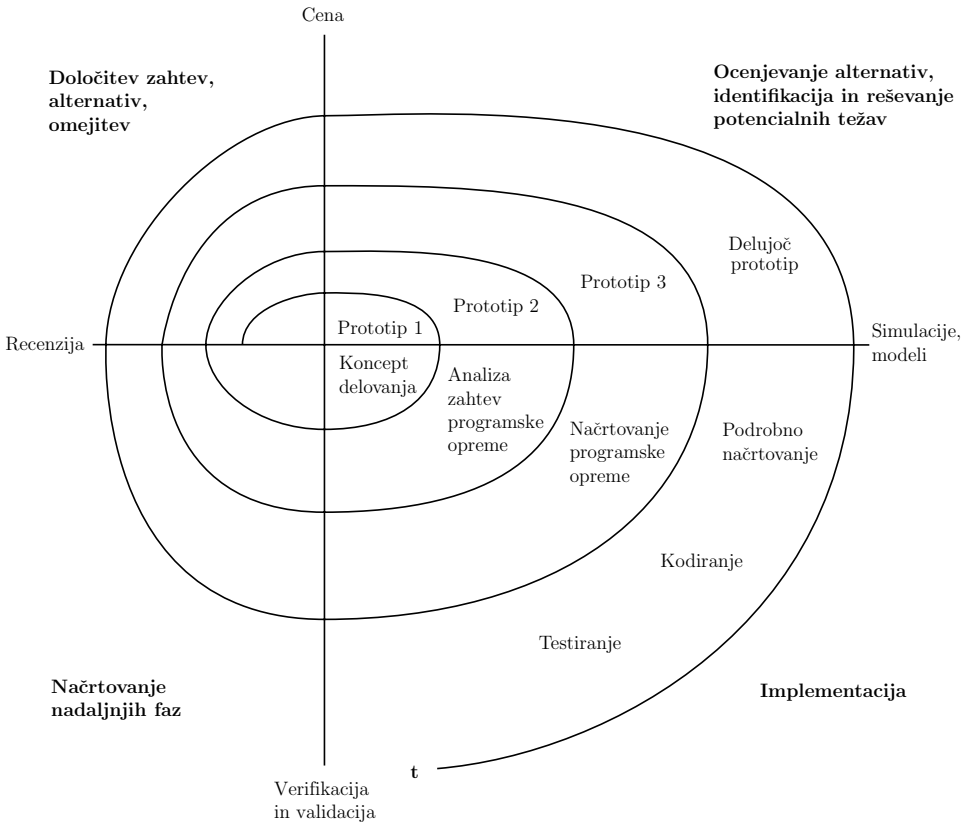
Slika 2.6: Korak v postopnem razvoju programske opreme

Pri razvoju s pomočjo prototipov smo že opisali možnost, da prototip postopno v določenem številu korakov postane končni produkt. Drugi možni način pa predstavlja postopek tako imenovanega postopnega razvoja programske opreme. Pri tem postopku razvoja se funkcionalnost produkta postopoma širi, med koraki postopnega širjenja funkcionalnosti pa se produkt že redno uporablja (slika 2.6). V vsakem koraku širjenja funkcionalnosti se uporablja klasični razvojni model. Na ta način programska oprema postopoma raste in se vse bolj prilagaja uporabnikovim zahtevam. Ta postopna metoda razvoja uporabnika prisili, da funkcije, ki jih od programske opreme pričakuje, razvrsti po pomembnosti. Če se gradi celotna programske opreme v enem koraku, se poleg res bistvenih funkcij na spisku želja pogosto znajdejo funkcije, ki niso res nujne ali ki se jih skoraj nikoli ne uporablja. Analitiki seveda težko ločijo funkcije po pomembnosti in tako lahko veliko razvojnega dela posvetimo delom sistema, ki sploh niso potrebni. Take preobsežne sisteme pa je zaradi kompleksnosti tudi težje uporabljati. Postopni razvoj programske opreme pa omogoča dodajanje le tistih funkcij, ki so potrebne in to takrat, ko postanejo potrebne. Taki sistemi so zato manjši in preglednejši. Gilb [18], eden glavnih zagovornikov postopnega razvoja programske opreme, meni, da je tak način razvoja tudi lažje obvladovati s projektnega vidika.

2.3.5 Spiralni razvoj programske opreme

V življenjskem ciklu programske opreme prvotnemu razvojnemu postopku sledi še vrsta vzdrževalnih posegov. Vsak vzdrževalni poseg je zopet razvojni cikel v malem, saj je potrebno opraviti analizo, načrtovanje, izvesti popravke v kodi, nakar še testirati. Tak vzdrževalni razvojni cikel je sicer napravljen na že obstoječem produktu, vendar ni bistveno drugačen kot osnovni razvojni cikel tega produkta. Tudi prvotni razvojni cikel običajno ne začnemo čisto od ničle, temveč upoštevamo že obstoječe sisteme in postopke. Zato lahko trdimo, da se vzdrževalni razvojni cikli od osnovnega po obsegu sicer razlikujejo, v svojem bistvu pa ne. Glavne razlika je to, da se pri vzdrževanju ponavadi veliko bolj mudi. Pri vzdrževalnih posegih kodo popravljamo in dodajamo ali krpamo, to pa postopoma slabša strukturo sistema. Zaradi pomanjkanja časa pogosto zanemarjamo še dokumentacijo, kar še bolj oteži nadaljnje vzdrževanje. Raziskovalci, ki so preučevali dinamiko evolucije programske opreme, so postavili v zvezi s tem nekaj zanimivih zakonov [35]:

Zakon stalnih sprememb. Sistem je smiselno vzdrževati toliko časa, dokler ga ni ekonomsko bolj upravičeno nadomestiti s povsem novim sis-



Slika 2.7: Spiralni razvoj programske opreme

temom.

Zakon naraščajoče kompleksnosti. Zaradi sprememb se programskim sistemom slabša struktura (entropija narašča) in zato postajajo vse bolj kompleksni. Za preprečevanje pretirane kompleksnosti je potrebno dodatno delo. Pri rasti programskih sistemov veljajo podobne zakonitosti kot pri socioekonomskih sistemih. V rasti mesta (progresivna aktivnost – gradnja novih cest, zgradb) se tudi ne sme zanemariti antiregresivnih aktivnosti (odvoz smeti, čiščenje, vzdrževanje obstoječih cest in zgradb), čeprav politično gledano ne prinašajo veliko ugleda. Če zanemarimo vzdrževanje, se to sicer ne pozna takoj, dolgoročno pa ogrozi tudi nadaljnjo rast.

Zakon evolucije programske opreme. Rast globalnih sistemskih atri-

butov je kratkoročno sicer lahko zelo hitra, dolgoročno pa se samoregulira in je skoraj linearna. Obdobju hitre rasti atributov (število vrstic, modulov, funkcij) nujno sledi obdobje, ko je potrebno kodo restrukturirati, ažurirati dokumentacijo, preden lahko sledi nov cikel rasti. Obe vrsti aktivnosti se tako dolgoročno izmenjujeta in tvorita stabilno kontrolno povratno zanko.

Če gledamo celotni življenjski cikel programske opreme, je razvoj prve verzije programske opreme le prvi korak. Klasični razvojni cikel daje statično sliko programske opreme tega prvega koraka. V resnici pa se programska oprema stalno razvija. Programska oprema zato ni zgrajena enkrat za vselej, ampak se stalno izpopolnjuje. Sliko programske opreme preko njenega celotnega življenjskega cikla daje spiralni model razvoja (slika 2.7).

Spiralni model razvoja programske opreme, kot že samo ime pove, vsebuje več ciklov, od katerih vsak vsebuje fazo analize, načrtovanja, implementacije in testiranja. Prvi od teh ciklov so namenjeni razvoju prototipov, kasnejši pa za adaptacijo obstoječih sistemov (vzdrževanje). Tudi cikel za osnovni razvoj sistema se lahko večkrat ponovi, prvič za del sistema, kasnejši cikli pa za druge dele sistema, podobno kot je to predvideno v modelu postopnega razvoja programskega opreme. Boehm [7], ki je predlagal spiralni model razvoja programske opreme, je vanj vključil vse do sedaj opisane modele razvoja:

- Če so zahteve nejasne, lahko sledimo spirali tolikokrat, da s pomočjo prototipov rešimo problem.
- Če so zahteve jasne in želimo zgraditi robusten in dobro dokumentiran sistem, sledimo spirali enkrat in uporabimo klasični razvojni cikel.
- Postopno lahko zgradimo sistem tako, da spiralo večkrat obkrožimo, vsakokrat za en del sistema.
- Vsaka napaka ali zahteva po spremembah med vzdrževanjem sproži nov obhod spirale.

2.4 Kakšna je razvojna stopnja programskega inženirstva?

Programsko inženirstvo še ni znanost v klasičnem smislu. Razvilo se je iz računalništva in pri delu uporablja številna formalna teoretična znanja (algoritmi, programski jeziki, prevajalniki, podatkovne strukture itd.). Na

drugi strani pa programsko inženirstvo potrebuje številna druga, manj formalizirana znanja, kot so vodenje in organiziranje razvojne skupine, ocenjevanje potrebnega dela, vprašanje prenosljivosti, robustnosti, prijaznosti do uporabnika in vzdrževanja programske opreme. Ta "mehkejša" znanja, ki so za uspešen razvoj programske opreme prav tako pomembna kot teoretični dosežki, so v obliki raznih navodil, postopkov, metod in orodij, razvitih v veliki meri na osnovi izkušenj. Zaradi te izkustvene narave razvoja programske opreme in relativne mladosti dejavnosti še ni enotne in splošne metodologije, na katero bi vsi prisegali. V literaturi so opisani številni metodološki pristopi [9, 47, 71, 18, 51, 65], v praksi pa mora vsaka razvojna skupina te postopke nadgraditi še s svojimi lastnimi izkušnjami. Večjo standardizacijo razvoja programske opreme postopoma prinašajo šele integrirana orodja za razvoj programske opreme (CASE – Computer Aided System Engineering), ki bodo vsilila razvijalcem določen način dela.

Programsko inženirstvo je še vedno nekje vmes med obrtniškim in industrijskim načinom izdelave. Nadzor razvojnega procesa v smislu doseganja načrtovane kvalitete, porabljenega časa in stroškov je še vedno slabši v primerjavi z drugimi zrelejšimi tehniškimi področji. Da bi dosegli boljši nadzor nad razvojnim procesom, bo treba preseči tradicionalno delitev na osnovni razvoj in nadaljne vzdrževanje. Če ista skupina ni zadolžena tudi za vzdrževanje, ni dovolj motivirana za razvoj take programske opreme, ki jo je enostavno vzdrževati. Če je skupina zadolžena za razvoj več podobnih programskih sistemov, je smiselno uvesti ponovno uporabo posameznih elementov programske opreme. Tako so novi produkti v družini sorodnih produktov zgrajeni iz že obstoječih polizdelkov. Na ta način se s pomočjo standardizacije, delitve dela, mehanizacije in avtomatizacije ter uporabe zamenljivih oziroma ponovno uporabljivih elementov počasi uvaja industrijski način izdelave programske opreme. V takem načinu razvoja se tudi vodenje ne sme osredotočiti le na posamezne časovno omejene projekte, marveč na kontinuirano skrb za življenski cikel cele družine produktov.

Poglavje 3

Projektno delo

Projekt je **enkratna,** praviloma **zahtevna** in **kompleksna** skupina nalog, ki mora biti končana v **določenem roku,** doseči mora vnaprej določene in morebitne kasneje odkrite **cilje** ter upoštevati vse podane in kasneje odkrite **omejitve.** Oglejmo si zdaj po vrsti bolj podrobno vse omenjene lastnosti projektov.

Enkratnost projekta

je v tem, da ne gre za ponavljajoč se proces, temveč za vsebinsko in časovno enkratno nalogo. Enkratnost projekta ne pomeni, da se projekt ne sme ponoviti v obdobju ene generacije. Enkratnost je v tem, da ne gre za ponavljajočo se nalogo, ki se jo izvaja trajno, ali tako, ki se ponavlja v znanih časovnih razmikih.

Zahtevnost projekta

je pogojena z zapleteno vsebino, pogosto pa tudi z delom velike skupine različnih strokovnjakov. Zahtevno vsebino projekta običajno obvladamo z manjšim številom vrhunskih strokovnjakov. Če pa je zahtevnost projekta pogojena z obsežnostjo nalog, je treba koordinirati delo velikega števila ljudi z različnimi strokovnimi profili, z različno motiviranostjo in različnimi izkušnjami. Zahtevnost projekta se kaže tudi v koordinaciji velikega števila različnih tehničnih pripomočkov in natančni kontroli stroškov projekta.

Projektni način dela se je razvil zaradi obsežnih, zapletenih in težavnih enkratnih del. Toda metodologija vodenja in načrtovanja projektov, ki je bila za to razvita, se danes uporablja tudi za bolj *učinkovito* vodenje preprostejših in enostavnejših del, ki bi jih sicer bilo moč obvladovati in voditi

z drugačnim načinom organiziranja.

Kompleksnost projekta

Projekt je sestavljen iz več delov, ki so na videz dokaj samostojni, v resnici pa so med seboj bolj ali manj povezani in odvisni. Zato celotne naloge ne moremo začeti reševati po njenih delih, vse dokler ne odkrijemo njene strukture, to je povezave med njenimi elementi in optimalnega zaporedja izgradnje teh elementov. Bolj ko je projekt strukturiran, več časa si moramo vzeti za to, da odkrijemo njegove temeljne podsisteme, s pomočjo katerih določimo vrstni red izgradnje celotnega projekta. Pri tem gre za uveljavitev znanega pristopa "od zgoraj navzdol", katerega cilj je odkriti strukturo sistema. Temu analitičnemu postopku sledi modularna gradnja posameznih podsistemov običajno po pravilu "od spodaj navzgor".

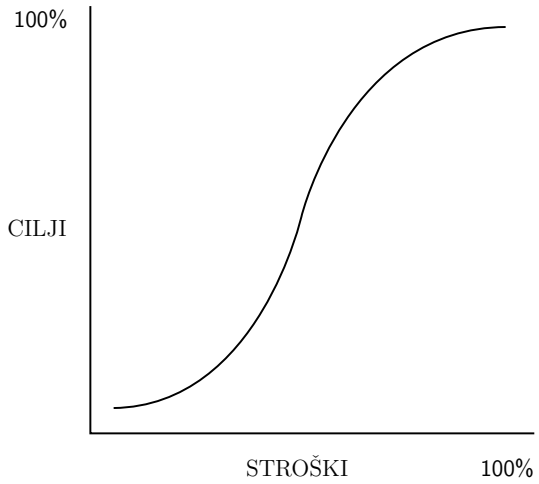
Rok projekta

Časovni termin ali rok, do katerega mora biti projekt končan, je odvisen od aktualnosti naloge, od razpoložljivih zmogljivosti za izvedbo projekta, pa tudi od motiviranosti podjetja. Ponavadi ni vseeno, kdaj bo projekt končan, bodisi zaradi zakonskih predpisov, tržnih razmer, konkurence, lahko pa tudi zaradi poslovne strategije podjetja. Če je postavljeni rok prekoračen, rešitev včasih sploh ni več potrebna.

Cilji projekta

Kadar govorimo o ciljih projekta, smo pozorni na tiste nove pridobitve, s katerimi bomo zadovoljili potrebe oziroma rešili probleme, ki so pogojili projekt. Pravilno izbrani cilji so jamstvo, da bodo ugotovljeni problemi ustrezno rešeni. Če ciljev ni mogoče določiti, če naročnik projekta zanje ne ve ali jih ne zna določiti, lahko rečemo, da naročnik ni pravilno izbran ali da projekt ni potreben. Če naročnik ciljev ne določi pisno ali če se mu to ne zdi potrebno (bodisi zato, ker jih ne zna določiti, ali ker to prepušča izvajalcu projekta), je bolje, da izvedbe take naloge ne sprejmemo. Cilji so pomembni za ocenjevanje napredka med delom pri projektu (presojamo, ali so doseženi delni cilji v skladu s končnimi cilji). Cilji so pomembni tudi za motivacijo, zlasti tedaj, ko pride do težav. Končno pa so cilji tudi merilo, s katerim ob koncu projekta naročnik in izvajalec presodita, v kolikšni meri je bil projekt uspešno končan.

Vseh ciljev ni možno vedno določiti na začetku projekta, zato mora imeti izvajalec projekta pravico in dolžnost, da ob delu odkriva dodatne oziroma



Slika 3.1: Stopnja in cena doseganja ciljev nista premo sorazmerni

nove cilje. Raziskovalni in razvojni projekti so celo vodeni tako, da se šele na osnovi delnih ciljev odločamo o nadaljnjem poteku projekta. O primernosti teh ciljev in njihovi vključitvi v projekt mora odločiti naročnik projekta. Pogosto ni smiselno, da vztrajamo pri uresničitvi vseh na začetku zastavljenih ciljev projekta. Odnos med stopnjo uresničitve ciljev projekta in ceno za dosego teh ciljev je prikazan na sliki 3.1. Kot je razvidno iz slike, je vztrajanje pri 100% izpolnitvi ciljev lahko združeno z nesorazmerno visokimi stroški, kar je skoraj vedno neracionalno.

Omejitve projekta

pomenijo, da izvajalci in tudi načrtovalci projektov nimajo popolne svobode. Pri svojem delu morajo upoštevati določena pravila, predpise, standarde, strojno in programsko opremo, navade, zgodovinska dejstva, jezikovno in splošno kulturo okolice, poslovno filozofijo in razpoložljivi kapital.

Med omejitve spadajo zlasti stroški za uresničitev projektov, razpoložljivi kadri za delo pri projektih, strojna in programska oprema. Nikomur ni vseeno, kolikšni bodo stroški razvoja projektov ter njihove kasnejše uvedbe in praktične uporabe. Med stroški in učinki projekta mora biti ustrezno razmerje—če so učinki nižji od stroškov, je bil projekt verjetno zgrešen. Temu se pri nas ne posveča potrebne pozornosti in natančnosti, saj bi sicer spoznali, da so mnogi na videz sicer odmevni projekti v resnici malo donosni in zato tudi neučinkoviti. Seveda je težko vnaprej oceniti vse možne učinke

saj so lahko učinki posredni, prikriti, skratka težko določljivi, a kljub temu odločilni za končno oceno.

Najpomembnejši faktor pri delu na projektih so prav gotovo ljudje s svojim znanjem, sposobnostmi in motiviranostjo.

3.1 Vrste projektov

Projekte lahko delimo po številnih kriterijih, na primer po namenu, objektu projekta, načinu izvedbe, trajanju projekta, kompleksnosti, lokaciji objekta, in vlogi projekta v kratkoročnem, srednjeročnem ali dolgoročnem razvoju podjetja.

Dve temeljni delitvi, ki odločilno vplivata na način vodenja in organiziranja projektov, pa sta delitev na *deterministične* in *stohastične* projekte ter na *enkratne* projekte in na projektne *procese.*

3.1.1 Deterministični in stohastični projekti

Deterministični projekti so tisti, kjer je moč končne cilje povsem determinirati ali določiti. S končnimi cilji so posredno določeni tudi delni cilji oziroma celotna struktura in izvajanje projekta. Za deterministične projekte je torej značilno *ciljno retrogradno* oblikovanje projektov. To pomeni, da se na osnovi jasno določenega končnega cilja postopoma določi vse aktivnosti, ki so potrebne za dosego tega cilja. Večina projektov, katerih cilji so uresničljivi s precejšnjo verjetnostjo, sodi v to skupino.

Stohastični projekti so tisti projekti, kjer končnih ciljev ni moč natančno definirati. To so največkrat raziskovalni in razvojni projekti, kjer šele delni rezultati začetnih aktivnosti omogočajo definicijo nadaljnjih ciljev. Tak postopni način oblikovanja projektov imenujemo *ciljno progresivni.*

3.1.2 Enkratni projekti in projektni procesi

Enkratni projekti se pojavljajo le enkrat. Vodenje takega projekta zato zahteva posebej zasnovano projektno organizacijo.

Projektni procesi pa so taki projekti, ki se v podobnih okoliščinah večkrat ponovijo. To so tipski projekti z enakimi ekonomskimi ali tehnološkimi značilnostmi. značilne primere takih tipskih projektov najdemo na primer v gradbeništvu. Ker zahtevajo nek ustaljen način izvedbe in vodenja, je njihovo vodenje zasnovano na stalni projektni organizaciji.

Različnim vrstam in načinom projektnega organiziranja je namenjeno naslednje poglavje.

3.2 Upravljanje in vodenje projektov

3.2.1 Razlika med projektnim in klasičnim upravljanjem

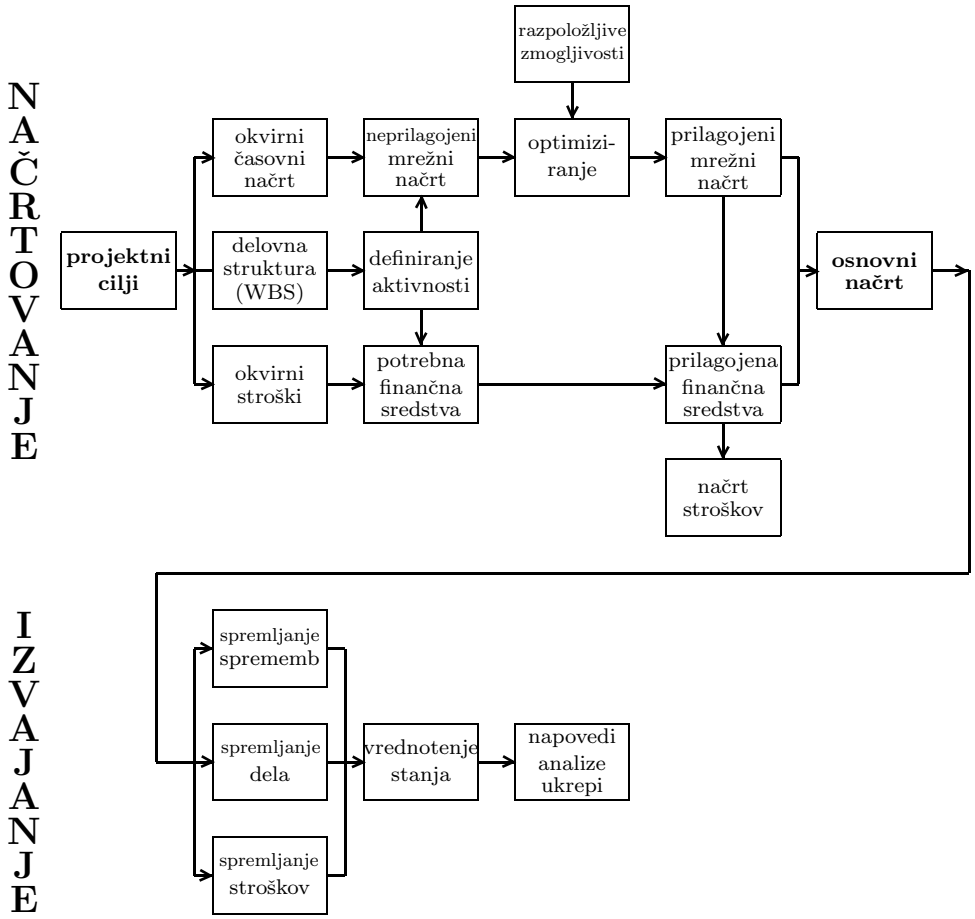
Upravljanje (angl. management) v klasičnem smislu zajema načrtovanje, organiziranje, kadrovanje, vodenje in kontroliranje virov oziroma sredstev nekega podjetja, da bi dosegli finančne in druge zadane cilje tega podjetja. V neprojektnem okolju se upravljanje koncentrira na produktivno, *časovno* orientirane kazalce uspešnosti. Upravljalci kontinuiranih dejavnosti se tako na primer sprašujejo: koliko izdelkov smo izdelali ta teden in koliko je to v primerjavi s prejšnjim tednom? Kakšna je bila prodaja in dobiček? Kakšna je bila izkoriščenost delovne sile in drugih zmogljivosti? Ali so stranke in zaposleni zadovoljni? Vsa omenjena vprašanja se nanašajo na dolgoročnost in kontinuiranost dejavnosti, zato se trenutna uspešnost primerja z enakimi kazalci v preteklih *časovnih* obdobjih.

Pri projektno orientiranem upravljanju ali vodenju projektov gre prav tako za načrtovanje, organiziranje, vodenje in kontroliranje virov oziroma sredstev, toda v *specifičnem časovnem obdobju* z jasnimi *enkratnimi cilji.* Čeprav se tudi v projektnem okolju lahko uporablja nekatere od zgoraj naštetih kazalcev uspešnosti na časovno enoto, pa je projektno vodenje strukturirano tako, da se lahko kontrolira uporaba virov in sredstev glede na *opravljeno delo,* ki je bilo definirano v okviru projektnih aktivnosti. Ta drugačen način gledanja se jasno zrcali tudi v drugačnih organizacijskih strukturah za projektno vodenje.

3.2.2 Vodenje projekta in spremljanje dejavnikov projekta

Projektno vodenje je večplasten proces, kjer hkrati spremljamo in upravljamo z:

- delom (aktivnostmi),
- časom,
- kadri,
- stroški,
- kvaliteto,
- komunikacijami.



Slika 3.2: Funkcije načrtovanja in izvajanja projektov

Vsakega od teh elementov je potrebno načrtovati, organizirati, spremljati in kontrolirati. Osnova za spremljanje in kontrolo je načrt, ki ga mora imeti vsak projekt.

3.3 Delovne faze projekta

Potek vsakega projekta lahko razdelimo na tri splošne faze:

1. *fazo zasnove* (določitev končnih in nekaterih delnih ciljev),

2. *fazo definiranja* (na osnovi zasnove se oblikuje projektni načrt) in

3. *fazo izvajanja* (izvajajo se načrtovane aktivnosti, da se dosežejo zastavljeni cilji).

Fazi zasnove in definiranja pogosto imenujemo kar na kratko *načrtovanje projekta.* Na sliki 3.2, kjer so definirane glavne funkcije načrtovanja in vodenja projektov, je prikazana delitev na fazo načrtovanja in izvajanja.

3.3.1 Faza zasnove projekta

V fazi zasnove projekta se izoblikujejo globalni *cilji* projekta. V tej fazi se opravijo razne pripravljalne študije, ki naj pripomorejo pri definiciji projektnih ciljev. Cilji morajo biti definirani glede na čas, sredstva, tehnične zahteve in podobno. Če je le mogoče, naj bodo ti cilji oblikovani kot jasno definirani *končni izdelki.* Končni izdelki se nanašajo na glavne komponente projekta. Za trženje novega produkta so na primer končni izdelki: embalaža, reklamiranje, poskusno trženje in na koncu še novi produkt. Za novo letalo bi taki končni izdelki lahko bili pogonski sistem, struktura in oblika, kontrolni sistem vodenja in tako dalje. V fazi zasnove moramo ugotoviti tudi, kako se cilji obravnavanega projekta vključujejo v splošno usmeritev organizacije, ki načrtuje projekt.

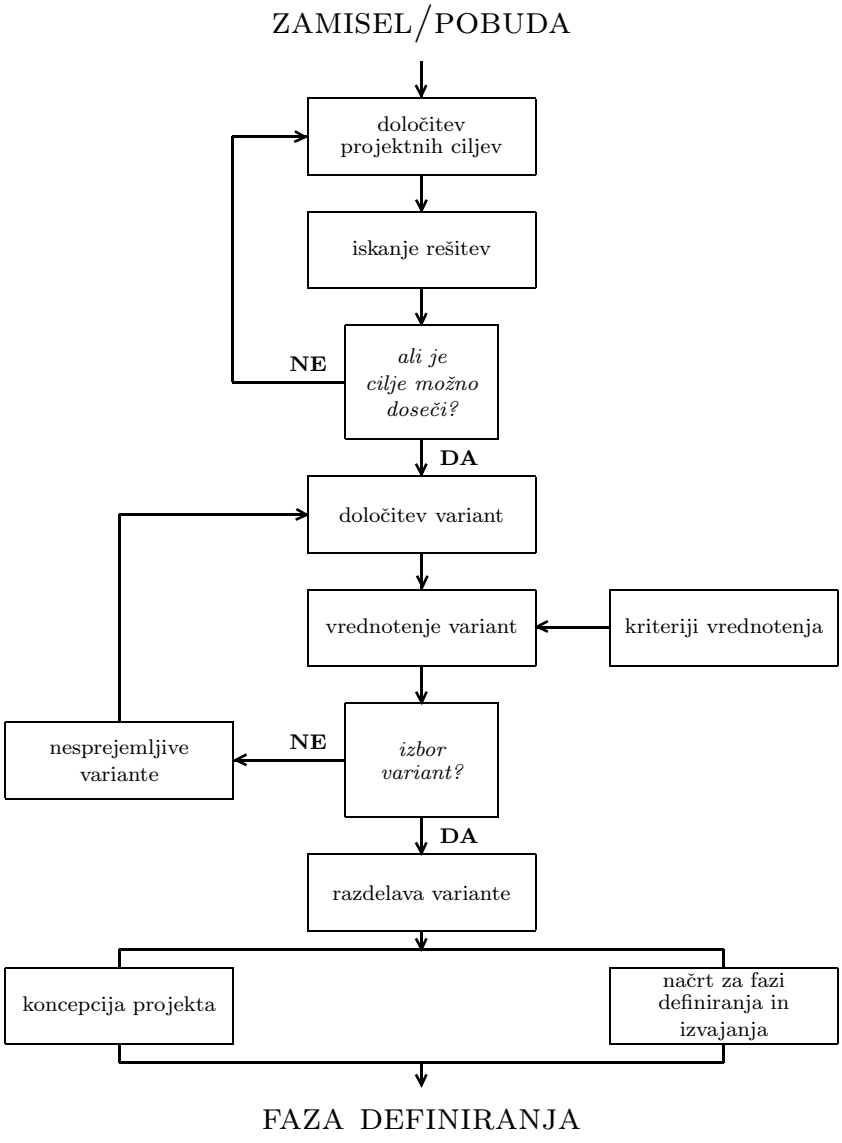
Potek faze koncipiranja je prikazan na sliki 3.3.

3.3.2 Faza definiranja projekta

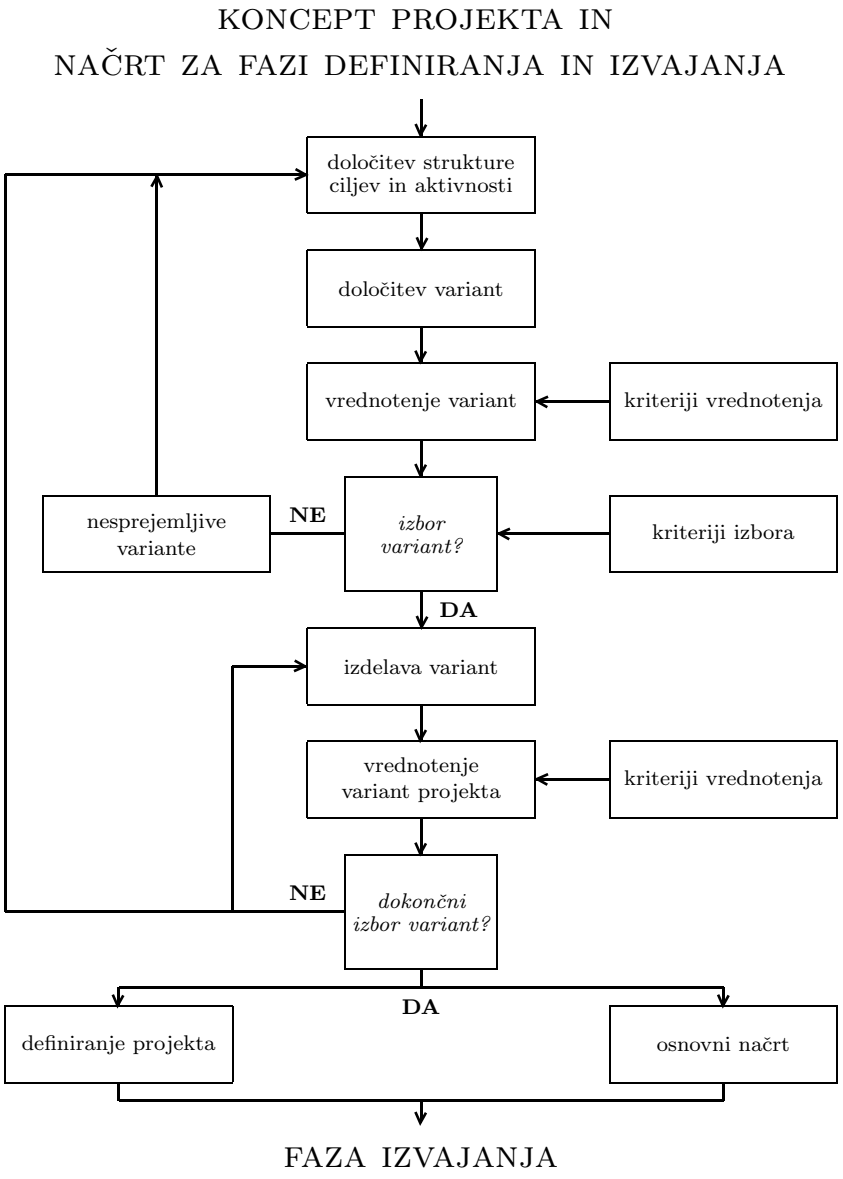
Na osnovi globalnih ciljev, ki so bili postavljeni v fazi zasnove, se v fazi definiranja projekta izdela *izvedbeni* ali *osnovni načrt,* ki je usklajen z vsemi danimi možnostmi in omejitvami. Kako poteka faza definiranja projekta, je prikazano na sliki 3.4.

Projekt mora biti določen *pisno.* Dokument, ki določa projekt, ni le osnova za primerjavo med naročenimi in uresničenimi rešitvami, temveč tudi podlaga za delo v projektni skupini. Med delom pri projektu se pogosto porajajo želje, ideje ali celo zahteve po razširitvi določene vsebine projekta — vanj naj bi vključili mejne probleme in tiste, ki so s projektom sicer povezani, niso pa direktno njegov sestavni del. Tako vključevanje novih in novih problemov v projekt ne pomeni samo širjenja projekta, temveč tudi zmanjševanje njegove obvladljivosti, hkrati pa obstaja velika nevarnost, da se projekt zruši zaradi preobsežnosti.

Točno določena vsebina projekta je torej pogoj za uspešen začetek dela, za reševanje vseh zapletov med delom pri projektu in ob izročitvi naročniku. Zato se pri zahtevnih projektih najprej izdelajo razne ekonomske predštudije. Čas, porabljen za čim bolj natančno določitev projekta, bo nekajkrat



Slika 3.3: Shema zasnove projekta



Slika 3.4: Shema definiranja projekta

povrnjen s tem, ker bo delo v naslednjih fazah potekalo bolj tekoče in z manj razmišljanja o tem, ali so "pota", po katerih se odvija delo na projektu, prava in usmerjena k pravim in naročenim ciljem. Vse kasnejše spremembe ponavadi precej podražijo in zavlečejo projekt.

V fazi definiranje se določi obseg dela, oceni potreben čas, ugotovijo razpoložljive in določijo potrebne zmogljivosti. Vse to je zajeto v tako imenovanem osnovnem načrtu, ki ga je potrebno še ovrednotiti, optimizirati in sprejeti.

Izdelava osnovnega načrta se začne na treh vzporednih poteh (glej sliko 3.2). Hkrati, toda v soodvisnosti, je potrebno oceniti delo, ki ga je potrebno opraviti, potreben čas in stroške za to delo. Uspeh celotne faze načrtovanja je odvisna prav od kvalitetne definicije potrebnega dela. Oglejmo si sedaj bolj podrobno vsebino teh funkcij:

Okvirni časovni načrt. Eden od namenov faze definiranja je tudi določitev splošnih časovnih okvirjev za izdelavo projekta. Ta časovni okvir se razdeli na mejnike za dosego glavnih podciljev projekta.

Določitev dela. Potrebno delo izrazimo s spiskom aktivnosti, ki so zaokrožene delovne naloge z jasnimi cilji. Med metodami za določanje dela prevladuje metoda, ki določa delovno strukturo (angl. Work Breakdown Structure — WBS).

Okvirni stroški. Že na samem začetku projekta moramo okvirno vedeti, koliko lahko za projekt sploh plačamo. Skoraj vsi projekti se morajo po nekem ekonomskem izračunu izplačati, to je, da se stroški projekta kasneje povrnejo, na primer v obliki novih, boljših ali hitrejših storitev programske opreme.

Izdelava mrežnega načrta. Z ozirom na okvirni časovni načrt in s pomočjo spiska aktivnosti in časovnim diagramom glavnih podciljev se začne izdelava mrežnega načrta. Za vsako aktivnost se mora oceniti potreben čas, nato pa določiti vrstni red izvajanja aktivnosti. Za mrežno načrtovanje poznamo več različnih metod in celo različnih vrst mrežnih diagramov. Eden od rezultatov mrežnega načrtovanja pa je določitev tistih aktivnosti, ki so odločilne za pravočasni zaključek projekta. Zaporedju takih aktivnosti pravimo *kritična pot*. Na tej poti so samo aktivnosti, ki nimajo časovne rezerve.

Za določitev potrebnih zmogljivosti je izhodišče spisek aktivnosti. Za vsako aktivnost posebej določamo potrebne kadre, opremo, storitve, skratka vse zmogljivosti, potrebne za dosego ciljev določene aktivnosti.

Pri tem upoštevamo okvirni načrt. Ker imamo že izdelan časovni potek vseh aktivnosti, lahko dobimo oceno skupnih potrebnih zmogljivosti za celoten projekt po posameznih časovnih enotah. Te potrebe lahko grafično prikažemo kot histograme zmogljivosti. V takem histogramu je moč hitro oceniti, kdaj potrebne zmogljivosti presežejo razpoložljive zmogljivosti.

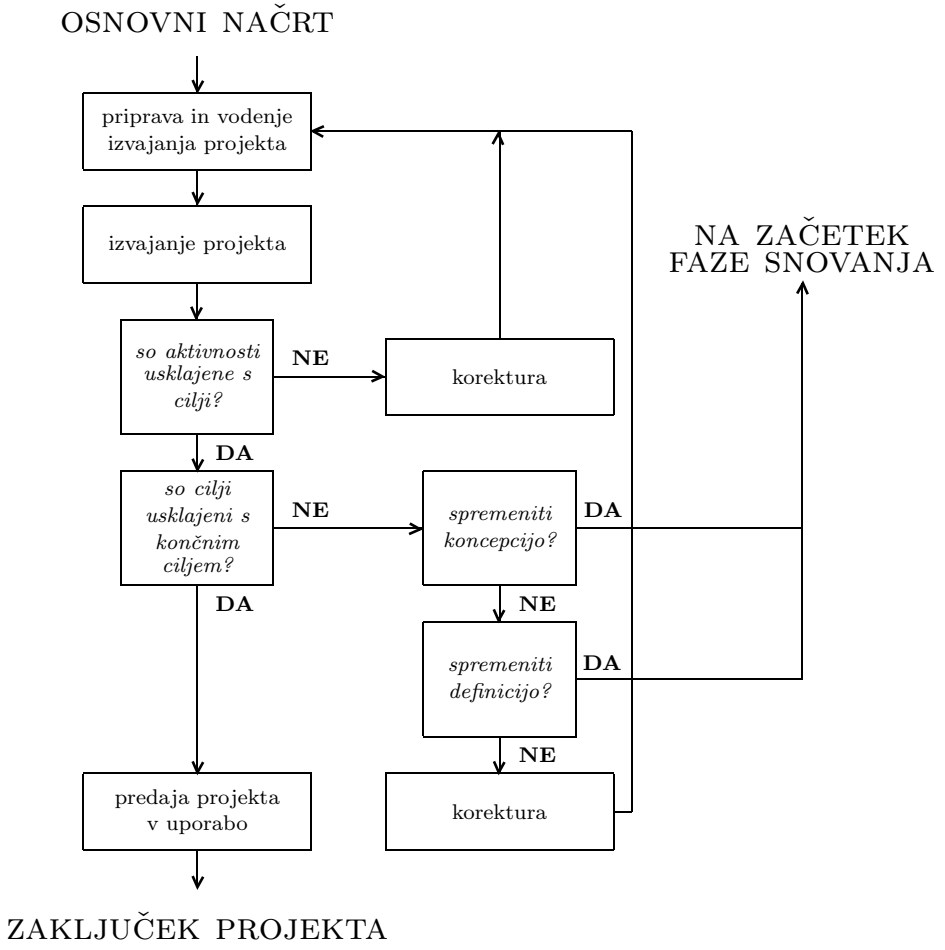
Ovrednotenje in optimiziranje mrežnega načrta. Vse predhodne funkcije so privedle do osnovnega mrežnega načrta projekta. Vprašanje pa je, če prva različica načrta tudi odgovarja zadanim časovnim okvirjem, stroškom in razpoložljivim zmogljivostim. Mrežni načrt je običajno potrebno ustrezno prilagoditi. Če ima histogram zmogljivosti izrazite vrhove in doline, je potrebno zmogljivosti izravnati. Pri tem pa je potrebno premakniti nekatere aktivnosti tako, da po možnosti ne podaljšamo izvajanje celotnega projekta. Izravnavanje zmogljivosti je del optimizacije osnovnega načrta.

Vrednotenje osnovnega načrta zajema tudi primerjavo s termini za dosego glavnih mejnikov projekta. Če se ti mejniki ne ujemajo z mrežnim načrtom, je morebiti potrebno dodati zmogljivosti posameznim aktivnostim. Računalnik je idealno orodje za analizo, ki je potrebna za optimiziranje osnovnega načrta. Računalnik izračuna najzgodnejše in najkasnejše možne začetke in konce vseh aktivnosti v mrežnem načrtu. Če je zaključek projekta prepozen, je možno vplivati na to s spreminjanjem številnih parametrov. Pri tem uporabljamo tako imenovano "KAJ-ČE" analizo, s katero lahko hitro preigramo različne možne scenarije. Optimizacija je iterativen proces, ki ga ponavljamo, dokler ne dosežemo dovolj dobrega kompromisa med cilji in danimi omejitvami.

Načrt stroškov. Stroške se najpogosteje določa na ravni aktivnosti, kjer so definirane tudi potrebne zmogljivosti. Določiti je potrebno ceno vsake zmogljivosti (ceno na časovno enoto in morebitne direktne stroške). Številni programi za podporo načrtovanja in vodenje projektov omogočajo kreiranje celotnega proračuna na osnovi spiska cen za posamezne vrste zmogljivosti.

3.3.3 Faza izvajanja projekta

Za spremljanje in kontrolo izvajanja projekta je važno, da imamo osnovni načrt, ki služi kot referenca za vse kasnejše spremembe. V fazi izvajanja



Slika 3.5: Shema izvajanja projekta

projekta se izvajajo aktivnosti, kot so definirane v osnovnem oziroma izvedbenem načrtu, ki je bil izdelan v fazi definiranja projekta. Izvajanje projekta shematsko prikazuje slika 3.5.

Osnovne funkcije načrtovanja in vodenja projektov, ki sodijo v to sklepno, izvedbeno fazo, so: spremljanje dela, spremljanje dejanskih stroškov, vrednotenje stanja ter napovedovanje, analiziranje in ukrepanje (glej sliko 3.2). Oglejmo si jih bolj podrobno:

Spremljanje dela. Spremljanje dela pomeni, da ugotavljamo, koliko dela je bilo dejansko narejenega, katere oziroma kolikšne zmogljivosti so

bile uporabljene in kakšni so bili stroški. Posebej se beleži, katere aktivnosti so bile začete in katere so bile dokončane. Meritve opravljenega dela se izvaja ob rednih časovnih intervalih. Trenutni status projekta se lahko izrazi glede na čas, potreben za dokončanje projekta, ali z odstotkom dokončanih del. Stanje projekta se ponavadi ugotavlja ob vnaprej določenih rokih z organiziranjem kontrolnih sestankov. Če so ti roki enakomerne časovne periode (npr. mesečne), govorimo o posnetkih stanja. Stanje projekta pa se običajno ugotavlja tudi, ko dosežemo važnejše podcilje ali mejnike projekta (angl. milestones).

Spremljanje dejanskih stroškov. Spremljanje dejanskih stroškov med izvajanjem projekta je važno zato, da lahko dovolj zgodaj odkrijemo negativne trende, ko dejanski stroški prehitevajo načrtovane.

Vrednotenje stanja. Vrednotenje izvajanja projekta ima cilj ugotoviti, ali je dejansko prišlo do odstopanja od načrta projekta. Pri vrednotenju izvajanja moramo sicer paziti na načrtovane začetke in konce aktivnosti, toda opazovati moramo tudi trende v daljših časovnih enotah. Negativni trendi v določenem časovnem obdobju so lahko le lokalne motnje, negativni trendi preko daljšega obdobja pa kličejo k ustreznim ukrepom.

Kontrola projekta sestoji iz več korakov, ki naj v končni fazi pripeljejo do potrebnih korektivnih akcij. Spremljati je potrebno predvsem datume dokončanja *aktivnosti na kritični poti*, pa tudi dokončanja drugih aktivnosti, saj če se le-te preveč zavlečejo, tudi lahko vplivajo na kritične datume dokončanja. Spremljanje opravljenega dela in dejanskih stroškov ter primerjava s predvidenimi stroški sta bistvena pri kontroli izvajanja projekta.

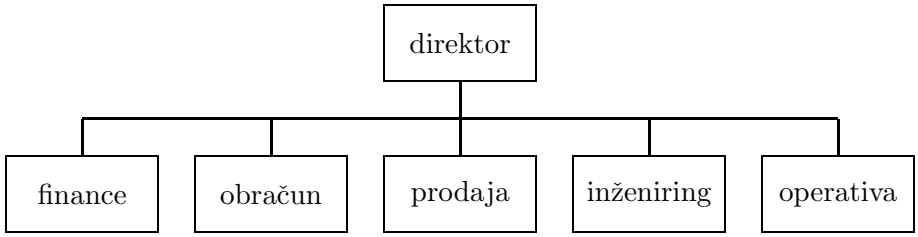
Napovedovanje, analiziranje in ukrepanje. Napovedovanje in predlaganje akcij je naravni podaljšek vrednotenja izvajanja. Cilj te dejavnosti je ugotoviti, *zakaj* je prišlo do motenj v izvajanju projekta in predlagati take korektivne akcije, da se zopet približamo osnovnim ciljem projekta. Da bi zadostili osnovnim ciljem, kot je na primer pravočasno dokončanje projekta, so včasih potrebni popravki osnovnega načrta. Napovedovanje se vrši na osnovi ekstrapolacije preteklih dosežkov. Za predlaganje korektivnih akcij je važno podrobno poznavanje vseh prej naštetih funkcij izvajanja projekta, da bi lahko izmed več alternativnih posegov izbrali najboljšega.

3.4 Organizacija skupin

Običajno so podjetja organizirana po funkcijskih načelih tako, da so posamezne organizacijske enote namenjene izpolnjevanju določenih funkcij. Čista projektna organizacija pa je postavljena za uresničitev povsem določenega projektnega cilja in je običajno razpuščena, ko je konec projekta. Sredi 60-tih let se je pojavila *matrična organizacija* kot kombinacija omenjenih dveh organizacijskih oblik in je rešila problem deljenja zmogljivosti in odgovornosti, pa kljub temu obdržala zadostno kontrolo nad funkcijami in projekti. Z matrično organizacijo se je še povečala potreba po skrbnem načrtovanju, kontroliranju in komuniciranju med udeleženci v tej organizacijski shemi.

3.4.1 Tradicionalna funkcijska organizacija

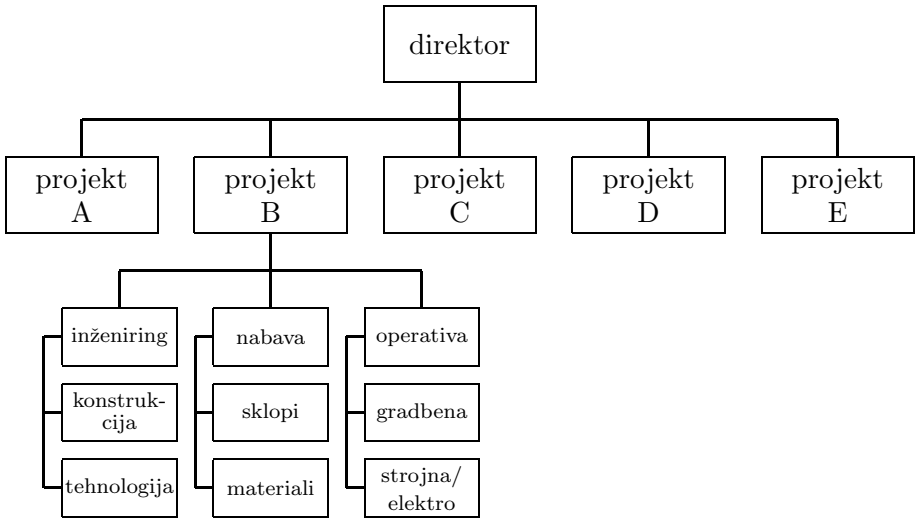
V funkcijsko bazirani organizaciji (slika 3.6) se upravljalci trudijo izboljšati tiste časovno orientirane kazalce uspešnosti, ki smo jih že omenili v razdelku 3.2. Uspešnost vodilnega osebja se ocenjuje po tem, kako dobro se posamezne funkcije izvajajo. Kadar so cilji posameznih funkcijskih oddelkov v nasprotju s cilji projektov, se daje prednost funkcijskim ciljem, kar je tudi naravno v funkcijsko bazirani organizaciji. Če posamezni projekti nimajo posebnega skrbnika, se morajo vsi problemi med različnimi funkcijskimi enotami razrešiti na nivoju vodstva podjetja.



Slika 3.6: Tradicionalno, po funkcijah organizirano podjetje

3.4.2 Čista projektna organizacija

Čista projektna organizacija (slika 3.7) je uveljavljena pri tistih podjetjih, ki vso svojo dejavnost izvajajo s projekti, predvsem z zelo velikimi. Njena glavna značilnost je, da so delavci razporejeni po posameznih projektih; ko pa se delo pri projektih konča, se “preselijo” k drugemu projektu ali pa so začasno razporejeni v funkcijsko enoto, da tako zapolnijo nastale časovne



Slika 3.7: Čista projektna organizacija

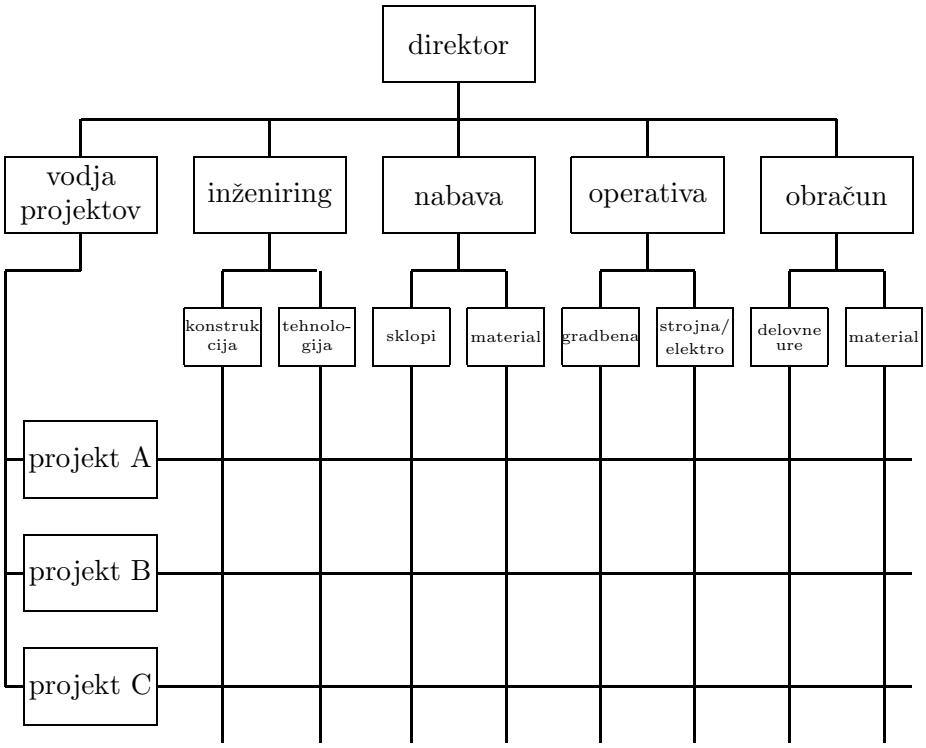
vrzeli med dvema projektoma. Najbolj razširjen primer takšne organiziranosti so razna gradbena in montažna podjetja, ki imajo poleg gradbišč organizirane tudi določene funkcijske obrate, kjer delavci najdejo "delovno zatočišče" za čas, ko niso razporejeni k nobenemu projektu.

Čista projektna organizacija, v kateri delavci po končanem projektu izgubijo delo, je uveljavljena predvsem v tistih državah, kjer se mora s socialno varnostjo ukvarjati delavec sam, ne pa družba ali država.

3.4.3 Projektno-matrična organizacija

Temeljna značilnost projektno-matrične organizacije (slika 3.8) je, da funkcijski organizacijski strukturi odvzame določene vloge; gre torej za porazdelitev vlog med funkcijsko in projektno organizacijo. Seveda pa ne zadošča zgolj natančna določitev vlog, temveč morajo biti vloge tudi sporazumno dogovorjene in kontrolirane pri njihovem praktičnem izvajanju.

Značilnost projektno-matrične organizacije je v tem, da so vsi viri znanja, izkušenj, sposobnosti *(kako, kdo)* in oprema praviloma v funkcijsko organiziranih enotah, medtem ko ima projektna organizacija več ali manj natančno opredeljeno nalogo z vsemi omejitvami in roki *(kaj, do kdaj)*. Čim bolj je projektna organizacija odvisna od kadrov v funkcijsko organiziranih enotah, tem slabši je njen položaj in s tem večjimi težavami se srečuje pri delu.



Slika 3.8: Matrično organizirano podjetje

Odnos do projektne organizacije tega tipa se kaže v tem, kateri delavci iz funkcijskih enot so razporejeni k posameznim projektom — če so to ustvarjalni, za delo zavzeti in motivirani delavci z bogatimi izkušnjami in obilico znanja, je to zanesljiv dokaz, da je projekt v središču zanimanja določene organizacijske enote. Povsem drugače (negativno ali nevtralno) pa je, če so za projekt določeni delavci, ki teh lastnosti nimajo oziroma imajo lastnosti, s katerimi ovirajo ali otežujejo delo pri projektu.

Osnovni pogoj za uspešno projektno-matrično organizacijo je pravočasno in natančno planiranje vseh aktivnosti projekta in vseh zmogljivosti, potrebnih za izvedbo. Ker so te zmogljivosti praviloma v različnih organizacijskih enotah, ki so v precejšnji meri zasedene še s svojimi rednimi (operativnimi) nalogami, je natančen načrt projekta osrednji dejavnik, ki omogoča, da ne prihaja do večjih težav, ovir ali zastojev pri reševanju projektnih nalog.

Dvojna podrejenost izvajalcev projekta (to je funkcijskemu vodji in vodji projekta) lahko povzroča težave in spore. Pri tem je pomembna stopnja

samostojnosti, ki jo imajo delavci, ki so razporejeni k projektu, v odnosu do svojega funkcijskega vodje. Če so med opravljanjem projektne naloge razbremenjeni rednega dela na svojem siceršnjem delovnem mestu in če lahko samostojno odločajo, kateri nalogi bodo dali prednost, odpade večina problemov, ki sicer povzročajo spore ali zastoje. Če pa je izvajanje projektne naloge za izvajalce drugotnega pomena ali če morajo pri projektnem delu v celoti upoštevati navodila, pričakovanja in smernice svojega funkcijskega vodje, je delo v projektno-matrični organizaciji izredno težavno in polno zapletov. Poseben problem pri uveljavljanju projektno-matrične organizacije je razmejitev odgovornosti funkcijskega in projektnega vodje pri ocenjevanju delovne uspešnosti delavcev, pri njihovem napredovanju, izobraževanju, pri izostankih z dela, dopustih in podobno. Če ta pooblastila niso natančno razmejena, tako situacijo lahko izrabijo tisti, ki se radi izmikajo vsakemu organiziranemu delu.

Med znanimi težavami projektno-matrične organizacije je tudi ohranjanje stika z matično enoto in delom, h kateremu je delavec sicer razporejen. Tu ne gre le za ohranjanje delovne "forme" in njegovega statusa, temveč predvsem za to, da ohrani stik pri vsebinsko najbolj zahtevnih opravilih. To lahko dosežemo tako, da je delavec ob razporeditvi k projektni nalogi oproščen dela pri rutinskih (ponavljajočih se) nalogah, zadrži pa vse naloge, bistvene za vsebino njegovega delovnega področja. Potrebno ravnovesje med funkcijami, ki jih je delavec opravljal na rednem delovnem mestu, in funkcijami, ki jih opravlja pri projektu, se doseže na primer tako, da sodeluje pri projektni nalogi največ tri dni na teden ali neprekinjeno teden dni, ves ostali čas pa mora opravljati delovne naloge na svojem rednem delovnem mestu. Za vsaj nekajdnevno strnjeno delo govorijo izkušnje, saj je delo pri projektni nalogi, ki traja manj kot 2 dni naenkrat, premalo učinkovito, ker se preveč časa porabi za "ogrevanje", tako da za reševanje projektne naloge ostane premalo časa. Zato tudi ni priporočljivo, da dela delavec na več projektih hkrati.

3.4.4 Skupina glavnega programerja

Skupina glavnega programerja je posebna organizacijska oblika za razvoj programske opreme, ki se zgleduje po kirurški ali pilotski ekipi. Tako kot kirurg ali pilot tudi glavni programer sam opravlja vse bistvene naloge, od analize, načrtovanja, celo kodiranja, drugi člani ekipe pa mu pri tem asistirajo. Glavni asistent ga lahko kratkoročno tudi nadomesti, tretji član skupine, to je knjižničar, pa skrbi za administracijo in dokumentacijo. Skupini se lahko (občasno) pridružijo tudi drugi eksperti. Tako elitistično zasno-

vana skupina se je izkazala za zelo učinkovito. Vendar pa je ljudi, ki lahko igrajo vlogo glavnega programerja, enostavno premalo, saj ta vloga zahteva poleg tehnične kompetentnosti še precej voditeljskih sposobnosti. Za večje projekte take majhne skupine tudi niso dovolj zmogljive. Za velike projekte se je zato uveljavila modificirana oblika, kjer v vlogi glavnega programerja nastopa skupina enakovrednih strokovnjakov, ki vodijo in nadzorujejo vse večje skupine programerjev. Vodilna skupina je kolektivno odgovorna za celoten projekt.

3.5 Vloge v organizaciji projektov

Tako kot funkcijsko-hierarhična organizacija ima tudi projektna organizacija funkcije generalnega *usmerjanja, upravljanja, vodenja* in *izvajanja* projektnih nalog. Vsaka od teh funkcij ima povsem specifično vlogo in svojevrsten način delovanja, zato lahko govorimo o hierarhični strukturi projektne organizacije. V projektni organizaciji morata biti najmanj dve vlogi, ki sta bistveni za projektno organiziranost, ki ju opravlja posebej organizirana skupina delavcev ali posameznik. *Vsak projekt* mora imeti svojega *naročnika* in *izvajalca,* lahko pa ima še strokovno-svetovalno ali revizijsko skupino.

Sistem vodenja obsežnih projektov je hierarhičen in ima naslednje tri nivoje:

- strateško planiranje,
- upravljalska kontrola,
- operativna kontrola.

Aktivnosti, ki sestavljajo mreže in ki ustrezajo tem trem nivojem vodenja morajo biti definirane tako, da dajejo posameznim nivojem vodenja ustrezne informacije za pregled nad stanjem projekta in sprejemanje ustreznih odločitev.

Generalni mrežni načrt sestavljajo aktivnosti, ki so ključnega pomena za strateški nadzor projekta. Če so za strateško načrtovanje potrebne podrobnejše informacije o posameznih aktivnostih, jih lahko dobimo iz podmrež 2. oziroma 3. nivoja, kjer so posamezne aktivnosti razdelane bolj podrobno, kolikor je pač potrebno za upravljanje oziroma izvajanje projekta.

Podjetja z močno razvejanim projektnim delom imajo poleg omenjenih struktur še generalnega upravljalca celotne projektne organizacije na najvišji upravljalski ravni.

3.5.1 Naročnik projekta

Vsak projekt mora imeti svojega naročnika, ki je praviloma edini ali večinski uporabnik projekta. Če je uporabnik projekta ena sama organizacija ali organizacijska enota, lahko kot naročnik ali kot investitor projekta nastopa vodja organizacijske enote. Pogosto pa se dogaja, da poleg večinskega uporabnika projekta nastopajo tudi druge organizacijske enote ali celo podjetja. V takih primerih je najbolje, da kot naročnik ne nastopa le večinski uporabnik, temveč tudi ostali uporabniki projekta. Obstaja namreč nevarnost, da se večinski uporabnik preveč vpleta v delo projektne skupine in v njene rešitve; zanimajo ga predvsem njegove lastne potrebe, zato kot naročnik skrbi predvsem za to, da bodo upoštevane zlasti njegove zahteve, medtem ko naj bi bile zahteve in potrebe drugih vključene v projekte le, če niso v nasprotju z njegovimi.

V vlogi naročnika praviloma nastopa posameznik ali projektni svet (odločitvena skupina), zlasti takrat, ko gre za projekt, ki posega v več poslovnih funkcij ali avtonomnih organizacij v okviru večjega podjetja. Vlogo naročnika lahko opravlja za vsak projekt posebej imenovana skupina ali skupina, imenovana za skupino sorodnih projektov, ki predstavljajo zaključeno celoto. Imenovanje skupine kot naročnika je nujno zlasti pri projektih, ki jih izvajajo v velikih podjetjih in ki so uvedeni v več organizacijskih enotah. Vodenje skupine v vlogi naročnika mora v takem primeru prevzeti oseba, ki je najbolj zainteresirana za to, da bi projekt v resnici zaživel.

Pri večjih in zelo zahtevnih projektih nastopajo tudi kombinirane naročniške skupine. Poleg najodgovornejših funkcionarjev podjetja je v delo skupine vključen tudi zunanji strokovnjak. Prednost kombinirane skupine je v tem, da prisotnost zunanjega strokovnjaka prinese "svež veter" in tako omogoča lažjo vključitev novih idej v projektno rešitev.

3.5.2 Izvajalec projekta

Najboljša oblika izvajalca projekta je *posebna projektna organizacijska enota,* v kateri so sistemizirana vsa delovna mesta, potrebna za uresničevanje projektov. Takšna organizacijska enota razvija svoj lastni sistem projektnega dela, vse aktivnosti lahko izvede sama ali preko delovnih skupin, ki jih vključi v projekt. Če pa v svojem sestavu nima vseh ustreznih strokovnjakov, prepusti določene aktivnosti drugim organizacijskim enotam ali izvajalcem zunaj podjetja.

Velikost projektne skupine je odvisna od zahtevnosti in obsežnosti projekta, vendar kaže upoštevati znano pravilo, da so manjše skupine učin-

kovitejše od velikih. Meja med velikimi in majhnimi skupinami je število 10. Iz izkušenj vemo, da skupine z več kot 7 člani porabijo preveč časa za medsebojno komuniciranje, premalo pa za reševanje konkretnih nalog[1]. Učinkovitost in ustvarjalno delo v skupini dosežemo z razvitim medsebojnim sodelovanjem, kar pa je lažje doseči v manjših skupinah.

Ključne naloge projektne skupine so:

- podrobna in skrbna seznanitev z vsebino projektne naloge, z vsemi njenimi danostmi (standardi, glavni projekt, skupne informacijske osnove itd.), omejitvami in dokumentacijo ter drugimi informacijskimi viri, določenimi v odločbi;
- analiza naročnikovih zahtev, želja in pričakovanj kot izhodišče za določitev problemov in pričakovanih rešitev;
- izdelava vsebinske strukture in podrobnejšega načrta projekta z rokovnikom in imeni nosilcev posameznih aktivnosti ter s predračunom stroškov celotnega projekta;
- sestava okvirnega modela (podatkov in procesov) novega informacijskega sistema na podlagi popisa in analize obstoječih procesov, postopkov ter informacijskih in materialnih tokov (idejni projekt);
- izdelava projekta, njegova izvedba in prenos v prakso;
- seznanitev in usposobitev izvajalcev, uporabnikov in vzdrževalcev za izvajanje in vzdrževanje projekta;
- izročitev projekta v vzdrževanje z vso potrebno dokumentacijo;
- izdelava ocene o uresničitvi postavljenih ciljev, rokov in stroškov;
- usklajevanje dela z naročnikom ter seznanjanje le-tega s stanjem projekta in z uresničevanjem zastavljenih planov.

Posebni tip izvajalca projekta predstavlja koordinator projekta, ki je le koordinator vseh izvajalcev projektnih aktivnosti. Izvajalci pa so lahko organizacijske enote v podjetju ali zunanji izvajalci. Ta model organiziranja projektov se pogosto uporablja tedaj, ko so izvajalci projektne naloge zunaj podjetja. Projektna skupina je tako sestavljena iz predstavnikov vseh izvajalcev, ki so bodisi predstavniki organizacijskih enot podjetja ali pa predstavniki zunanjih podjetij, ki sodelujejo pri projektu. Vloga projektnega koordinatorja je predvsem v tem, da skupaj z vsemi predstavniki izvajalcev projekta izdela rokovnik projekta, da spremlja uresničevanje vseh aktivnosti, rešuje zaplete in organizira prenos projekta v prakso. Projektna

[1] Glej str. 50

skupina izvaja samo tiste aktivnosti, ki so povezane z vodenjem (planiranje, spremljanje, kontroliranje), medtem ko so dejanski izvajalci projekta zunanja podjetja ali druge "notranje" organizacijske enote.

Vodja projektne skupine

Osrednja osebnost v projektni organizaciji je vodja projekta. Čim bolj celovit, obsežen in zapleten je projekt, tem bolj zahtevna, tvegana in odgovorna je njegova naloga. Zato ni dovolj, če projekt strokovno obvlada, ampak mora biti tudi odličen organizator in poznavalec skupinske dinamike. V projektni skupini se namreč zelo intenzivno odvijajo različni psihosocialni procesi in procesi učenja. Nemalokdaj so prav ti procesi bolj zapleteni kot samo reševanje projektnih nalog.

Vodja projekta mora biti vsestranska oseba. Popolnega projektnega vodjo je težko najti, zato naj bi bile želene lastnosti predvsem vodilo pri izbiri najprimernejšega projektnega vodje. Vodja projekta, ki ni posebej usposobljen za to funkcijo in ki se s projektnimi nalogami poklicno ne ukvarja, je le redko lahko uspešen. Več o vodenju in organizaciji skupin je opisano v 4. poglavju.

3.5.3 Svetovanje in preverjanje

Pri zahtevnih projektih se k razvijanju in uresničevanju projekta pritegne tudi posebno svetovalno skupino, katere naloga je strokovna kritika in pomoč pri uresničevanju projekta. Predstavljajo jo strokovnjaki in izkušeni praktiki, ki poznajo prakso in imajo predstavo o potrebah v prihodnosti. Njihova naloga je predvsem ta, da kritično preverjajo ponujene rešitve in njihovo ustreznost ter predlagajo izboljšave ali dopolnitve. Kot izkušeni praktiki in teoretiki imajo svetovalci tudi velik vpliv na ustvarjanje projektu naklonjenega razpoloženja v okoljih, kjer se bodo rezultati projekta uporabljali. Sestava skupine in njena velikost je odvisna od zapletenosti in zahtevnosti projekta. Zaželeno je, da svetovalna skupina ni večja od 15–20 članov.

Svetovalna skupina se vključuje v projektno delo predvsem pri projektih, ki močno spreminjajo obstoječo tehnologijo dela, organizacijsko strukturo ter kadrovsko zasedbo. Pri vsebinsko zelo zahtevnih projektih se v delo na projektih vključi tudi interno ali zunanjo revizijo ali recenzente.

Poglavje 4

Psihološki in sociološki vidiki projektnega dela

Projektno delo je skupinsko delo, ki je povrhu tega ponavadi še komunikacijsko zelo intenzivno. Zato morajo vsaj vodilni člani projektnih skupin poznati osnovne psihološke in sociološke značilnosti dela v skupinah [59]. Pri projektih v tujini in v drugih kulturnih sredinah je za uspešno delo prav tako važno upoštevati lokalne običaje in navade [19].

4.1 Splošno o skupinah

Skupina je socialna združba določenega števila posameznikov z istimi potrebami in interesi, ki jih lahko zadovoljijo z uresničevanjem skupnih ciljev. Zato obstajata v skupini močna kooperacija ter interakcija, uveljavljen je sistem moralnih vrednot in norm, ki urejajo obnašanje posameznih članov, zlasti delovanje skupine kot celote. Iz teorije psihologije [68, 22] vemo, da so za delo v skupinah odločilni trije pogoji:

1. Interesi skupine morajo biti povezani — to je pogoj za močno kooperacijo in interakcijo.
2. Člani skupine si morajo prizadevati za dosego skupno določenega ali izbranega cilja.
3. V skupini mora biti močna osebna (notranja) in zunanja (medosebna) motivacija.

Ti pogoji morajo biti izpolnjeni v vsaki skupini, ne glede na njeno velikost, cilje in medsebojne odnose, razlika je le v njihovi intenzivnosti.

4.2 Vrste skupin

Skupine razvrščamo v razne tipe skupin po različnih merilih, kot so intenzivnost medsebojnih odnosov, velikost, način nastanka, vloga in cilji skupine.

V nadaljevanju tega podpoglavja bodo prikazane predvsem tiste vrste skupin, ki so za projektni način dela najbolj pomembne.

4.2.1 Primarne in sekundarne skupine

Po intenzivnosti medsebojnih odnosov razlikujemo primarne in sekundarne skupine. Primarne skupine so tiste z zelo intimno in intenzivno medsebojno interakcijo, z veliko čustveno povezanostjo, razvitim občutkom za solidarnost in povezanost s skupino, z velikim vplivom skupine na obnašanje in ravnanje posameznih članov. *Primarne skupine* so družina, zakonska skupnost in prijateljske skupine, včasih pa tudi šolski razredi in manjše vojaške enote. Primarne skupine so praviloma zelo majhne, kar pa ne pomeni, da je vsaka manjša skupina tudi že primarna. Velikost skupine je torej pomembno, ne pa odločujoče merilo za določitev statusa primarne skupine.

Delovne skupine z manj intenzivno interakcijo, z manjšo čustveno navezanostjo, z manj izrazitim občutkom pripadnosti skupini in z manjšim spoštovanjem skupinskih norm so sekundarne skupine. *Sekundarne skupine* so razni odbori, komisije ter skupine za reševanje problemov in izdelavo rešitev. Sekundarne skupine so tudi vse velike skupine.

4.2.2 Neformalne in formalne skupine

Formalne in neformalne skupine se razlikujejo po tem, v kolikšni meri si člani skupine sami izbirajo svoje cilje in določajo vloge posameznih članov ter koliko lahko sami vplivajo na delovanje skupine. Skupine, v katerih so te dejavnosti prepuščene članom, so neformalne, medtem ko so formalne skupine tiste, kjer so ta razmerja in cilji določeni v odločbah, pravilnikih, poslovnikih in statutih.

Za **neformalne skupine** je značilno, da si člani sami izbirajo in zastavljajo cilje, določajo svoje vloge, aktivnosti in položaje. Odnosi med člani skupine so dokaj osebni in neformalni, delovanje skupine pa se neprestano prilagaja potrebam svojih članov. Zato so vloge, dejavnosti in položaji posameznikov spremenljivi in se menjajo po potrebi. Tudi članstvo v skupini ni točno določeno, to velja zlasti za večje skupine. Značilnost neformalnih skupin je torej velika dinamičnost in svoboda delovanja, ki je ne utesnjujejo različne formalne zapreke in postopki. Neformalne skupine ne obstajajo

samo kot samostojne skupine, temveč pogosto nastajajo tudi znotraj drugih, zlasti formalnih skupin kot prijateljske skupine, skupine simpatizerjev in zaveznikov. Neformalne skupine praviloma nastajajo znotraj vseh formalnih skupin, vprašanje je le, kakšni so njihovi cilji in načini delovanja — lahko so pozitivni in ustvarjalni, torej podpirajo delovanje formalnih skupin, lahko pa so negativni in razdiralni in prizadevanja formalnih skupin izničujejo ali podirajo.

Neformalne skupine nastanejo znotraj formalnih skupin zato, da bi posamezniki lažje zadovoljili svoje osebne potrebe. Člani neformalnih skupin so med seboj tesneje povezani ne le med delom v skupini, temveč tudi v prostem času. Izkušnje kažejo, da so neformalne skupine običajno sestavljene iz 2 do 3 članov, včasih pa tudi iz več. Koristne so predvsem zato, ker dajejo članom večje zadovoljstvo in povečujejo produktivnost dela v skupinah. V formalnih skupinah, v katerih ni neformalnih skupin, prihaja do večjih medsebojnih napetosti, saj je identifikacija posameznikov s cilji skupine in s skupino šibka. V neformalnih skupinah prihaja do sproščene izmenjave mnenj ter do vzajemnega priznavanja pobud in idej. Preko neformalnih skupin se da najlaže vplivati na aktivnost formalne skupine. Neformalne skupine ne nastajajo vedno le iz pozitivnih nagibov, ki podpirajo delovanje skupine. Kadar ima neformalna skupina posebne interese, ki so v nasprotju z interesi in potrebami formalne skupine, jo imenujemo klika ali klan. Klike povzročajo napetosti in spopade med člani neformalne in formalne skupine ali med posameznimi neformalnimi skupinami znotraj ene formalne skupine. Nasploh lahko trdimo, da so neformalne skupine tisto gibalo, ki podpira skupinsko delo, hkrati pa posameznikom nudi večje zadovoljstvo ob uresničevanju svojih, predvsem socialnih potreb.

V **formalnih skupinah** so cilji in vsebina skupine določeni od zunaj, pogosto tudi mimo njenih interesov. Člani se vključujejo v skupino prostovoljno, pod pritiskom ali proti svoji volji. Za formalne skupine je značilno, da so vloge njenih članov točno določene, tako da jih le-ti ne morejo bistveno spremeniti. Člani skupine so ponavadi imenovani z odločbo, sklepom ali odlokom. V večjih skupinah običajno obstajajo določena pravila o tem, kdo lahko postane ali kdo mora postati član skupine; določeni so tudi vodja, njegov namestnik in drugi funkcionarji, odločilni za učinkovito delo formalne skupine. Uspeh skupine se ponavadi pripisuje vodstvu, neuspeh pa članom skupine. Nekatere formalne skupine pristajajo na določene rituale in izbirajo za svojega vodjo osebo, ki ima določene karizmatične ali druge posebne lastnosti. Med formalne skupine spadajo poleg strokovnih, projektnih in delovnih skupin, odborov ter komisij tudi podjetja, društva, klubi kakor

tudi večje družbene organizacije, kot so politične stranke, etične in verske skupine, narodi in države. V okviru vodenja skupin pa nas zanima predvsem ena vrsta formalnih skupin, to so delovne skupine.

4.2.3 Delovne skupine

Za nas sta pomembna dva tipa delovnih skupin, to sta proizvodna skupina in skupina za reševanje problemov.

Proizvodna skupina je manjša skupina, katere osnovna naloga so pretežno rutinska opravila za izdelavo določenih izdelkov ali storitev. Interakcija med člani skupine temelji predvsem na koordinaciji, ki je povezana z izvrševanjem naloge, s proizvodnjo določenih izdelkov ali storitev. Čeprav je izvajanje določenih proizvodnih nalog glavni motiv za delo v skupini, skušajo ljudje ob svojem delu zadovoljiti tudi druge potrebe, zlasti potrebo po osebni uveljavitvi, samopotrjevanju in dokazovanju. Socialni odnosi znotraj delovne skupine so običajno manj intimni in manj trajni, odvisni pa so predvsem od prostorske bližine članov skupine. Čim bližje so si člani skupine med delom in bolj ko je intenzivna slušna in vidna komunikacija med njimi, tem trdnejša je tudi skupina v psihosocialnem smislu. Skupine v večjem prostoru med seboj težje komunicirajo, zato praviloma razpadejo na več manjših neformalnih psiho-socialnih skupin.

Delovne skupine prištevamo med formalne skupine zato, ker so njihove naloge, pravice, pristojnosti in odnosi točno določeni. Vsi vedo, kdo je njihov vodja, kdo kontrolira njihovo delo in kakšen je njihov položaj v proizvodnem procesu.

Skupine za reševanje problemov so običajno ustvarjalne skupine, ki so lahko sorazmerno trajne (npr. upravni in izvršilni odbori, sekretariati v podjetjih in drugih organizacijah), praviloma pa so začasne (npr. komisije, projektne in delovne skupine) in so ustanovljene za uresničevanje povsem konkretne naloge. Med skupine za reševanje problemov uvrščamo tudi skupine raziskovalcev, ki delajo pri istem znanstvenem projektu, in razne strokovne skupine za odkrivanje ali uvajanje novih rešitev, četudi imajo določene značilnosti proizvodnih skupin.

Cilj skupine za reševanje problemov je predvsem uresničitev določenih nalog, ki so povezane s problemom, zaradi katerega je bila skupina ustanovljena. Značilnost teh skupin je v tem, da rešujejo naloge na neformaliziran, po potrebi na povsem nov način, z močno medosebno komunikacijo, z velikim tveganjem in z veliko odgovornostjo. Če je razprava temeljni način iskanja rešitve posameznih problemov, imenujejo take skupine za reševanje problemov diskusijske skupine. Čeprav je pri diskusijskih skupinah poudarek

na verbalni komunikaciji, ima tudi neverbalna komunikacija zelo pomembno vlogo. Tisti, ki za to obliko komuniciranja niso dovolj občutljivi, so prikrajšani za mnoge signale, ki kažejo soglašanje ali nasprotovanje sporočilom iz verbalne komunikacije.

Ker imajo skupine za reševanje problemov izbranega ali imenovanega vodjo, za ključne aktivnosti pa se zahtevajo določene strokovne kvalifikacije, so to tipične formalne skupine. To pa seveda ne pomeni, da posamezni člani skupine ne prevzemajo različnih vlog med reševanjem problemov. Za dodeljevanje posameznih vlog sta odločilna znanje in motivacija posameznika za uspešno rešitev naloge. Motivi za angažiranje posameznikov so zelo različni. Nekateri bi radi z rešitvijo problema pridobili dominanten položaj ali vpliv v skupini, drugim pa gre za osebni prestiž in tekmovanje. Motiviranost posameznikov se lahko izredno poveča, kadar gre za uresničitev njihovih idej ali idej inštitucije, ki jo zastopajo, če se z njo močno identificirajo. Poleg čistih produktivnih vlog so v skupini tudi posamezniki, ki prevzemajo druge socialne vloge, kot so vnašanje dobre volje v skupino, pomoč pri reševanju nasprotij in sporov in iskanje orientacije, kadar skupina zaide v krizo. Značilno za skupine za reševanje problemov je, da se znotraj njih formirajo tudi neformalne skupine ali koalicije in da med člani skupine v času, ko se ne ukvarjajo z nalogo, običajno ni tesnejših stikov.

Status posameznika v skupini ni opredeljen samo z njegovimi osebnimi prizadevanji in aktivnostjo v skupini, temveč tudi s statusom, ki ga prinaša s seboj v skupino. Zato se v skupini posveča večja pozornost idejam, predlogom, zamislim tistih oseb, ki imajo višji status, kar pa lahko negativno vpliva na vzdušje in odnose v skupini. Poseben položaj in vpliv na uspešnost skupin za reševanje problemov ima vodja skupine. Vodja ima formalno pravico in dolžnost, da usmerja in nadzira delo skupine, da vodi razprave in da tudi sicer vpliva na sprejem rešitev. Od njega v največji meri zavisi, ali bodo sprejete rešitve, ki so najbolj smotrne, ali vsiljene tiste, ki jih je predlagal človek z najvišjim statusom v skupini. Glavna naloga vodje skupine je, da vse strokovne pobude, predloge in rešitve enakopravno obravnava in da zagotovi v skupini demokratično in sproščeno ozračje. Vsakomur mora zagotoviti pravico, da odkrito in brez zadržkov pove svoje mišljenje, da preprečuje napade na posameznike in da skrbi za usmerjeno in učinkovito delo skupine.

Skupine za reševanje problemov so najbolj učinkovite, če v njih sodeluje manjše število članov. Raziskave kažejo, da so včasih skupine, manjše od 5 članov, premajhne, skupine z več kot 5 člani pa pogosto že prevelike. Zanesljive meje torej ni, ker je velikost skupine pač odvisna od problema, ki

Velikost skupine	Udeležba posameznih članov pri komunikaciji					
	1.	2.	3.	4.	5.	6.–10.
dvočlanska	58%	42%				
tričlanska	42%	34%	24%			
štiričlanska	37%	27%	21%	15%		
petčlanska	39%	-	-	-	9%	
6- do 10-članska	50%	-	-	-	-	2%

Tabela 4.1: Udeležba posameznikov pri komuniciranju v skupini, kjer 100% predstavlja ves razpoložljivi čas za komuniciranje v skupini.

ga skupina rešuje, od znanja in motiviranosti članov skupine. Nasploh velja pravilo, da so manjše skupine bolj učinkovite od večjih. V naslednjih dveh razdelkih si oglejmo osnovne značilnosti malih in velikih skupin.

4.2.4 Male skupine

Male skupine so tiste, ki imajo omejeno število članov. O tem, kje je meja med malimi in velikimi skupinami, si strokovnjaki niso povsem enotni. Tako nekateri psihologi menijo, da so male skupine tiste z razponom med 2 in 20 člani, vendar naj bi pri neformalnih in intimnih skupinah to število ne presegalo 8 do 10 članov. Najboljši odnosi se vzpostavljajo v skupinah, ki imajo 2 do 5 članov. Ne glede na vse razlike, ki obstajajo v mnenjih strokovnjakov, je na splošno osvojeno pravilo, da so skupine z manj kot 10 člani male skupine, z več kot 20 pa velike skupine.

Čim manjša je skupina, tem bolj lahko posameznik zadovoljuje svoje potrebe in tem večje možnosti ima za uveljavitev, priznanje, spoštovanje in samopotrjevanje. Občutek zadovoljstva zaradi skupinske pripadnosti je v malih skupinah neprimerno bolj razvit kot v večjih skupinah, saj je prispevek posameznika k skupni rešitvi naloge bolj opazen, neposreden in večji kot v veliki skupini. Tudi medosebna povezanost v malih skupinah je neprimerno večja kot v velikih skupinah, zato je tudi odsotnost z dela manjša.

Manjše skupine so učinkovitejše tudi zaradi obremenitve, ki jo povzroča komuniciranje v skupini. Če naj bo interakcija zelo intenzivna, mora imeti vsak član skupine možnost, da se vključi v komuniciranje. Z večanjem števila članov v skupini se povečuje tudi število komunikacijskih povezav ali kanalov. Tako na primer obstaja v skupini, ki jo sestavljajo 4 člani, 6 komunikacijskih kanalov. V skupini z 8 člani se to število poveča na 28, pri 12 članih pa že na 66 kanalov itd. S porastom števila komunikacijskih kanalov

pa se zmanjšuje možnost posameznika za komuniciranje [56]. Raziskave o povprečni udeležbi posameznika pri komuniciranju v odvisnosti od velikosti skupine so dale rezultate, podane v razpredelnici 4.1. V dvočlanski skupini porabi prvi član 58%, drugi pa 42% skupnega časa za komuniciranje. Več ko je članov v skupini, manj časa za komuniciranje odpade na posameznika. Razpoložljivi čas pa se ne razdeli enakomerno na vse člane skupine. Eni člani začnejo prevladovati, drugi pa komaj pridejo do besede. Pri petčlanski skupini je razpon med najbolj zavzetim in najmanj zavzetim govornikom že 39% : 9%, pri večjih skupinah pa je to razmerje še bolj neugodno. Iz vseh teh ugotovitev lahko sklepamo, da večje skupine prinašajo članom manj zadovoljstva, otežujejo komuniciranje in vpliv posameznika na skupne rešitve je neprimerno manjši. Kadar gre za reševanje intelektualnih in zahtevnih nalog, izkušnje kažejo, da je najbolj primerno število članov v skupini 5, sicer pa je to število lahko nekoliko večje ali manjše.

Negativne posledice se pojavijo tudi, če so skupine premajhne. V takih skupinah prihaja pogosto do večjih nesoglasij ali do stalne in pretirane obzirnosti, tolerantnosti in celo do omejevanja svobode. Iz praktičnih izkušenj in tudi teoretičnih raziskav je ugotovljeno, da skupina, ki šteje 2 člana, ni najbolj primerna, zlasti ne za strokovno delo. Zato je najmanjša skupina, pri kateri še pridejo do izraza vse značilnosti skupinskega dela tista, ki ima najmanj 3 člane. To pravilo velja upoštevati zlasti pri sestavi delovnih skupin, ki delujejo v okviru projektnih skupin. Priporočljivo je, da je število članov v skupini neparno, ker pri parnem številu prihaja praviloma do večjih napetosti, sporazum ali soglasje pa se težje doseže.

4.2.5 Velike skupine

V teoriji se šteje za velike skupine tiste z več kot 20 člani. Kot smo že omenili, ta meja ni točno določena, ker zavisi od značaja dela, od ciljev skupine in od motivacije članov za sodelovanje v skupini. Značilno za velike skupine je, da so to formalne skupine, kjer so precej natančno opredeljene vloge, cilji, pogoji za sprejem v skupino, pravila obnašanja itd. V skupini se ve, kdo je dejanski in kdo je formalni voditelj, kdo je moderator, kdo so glavni govorci, pritrjevalci, oponenti in kateri del skupine pripada eliti. Interakcija v skupini je praviloma slaba, članstvo največkrat ni točno opredeljeno, identifikacija s cilji skupine je relativno slaba, slaba pa sta tudi kontrola nad delovanjem članov skupine in nad uresničenjem njihovih interesov. Razlogi za včlanitev v velike skupine so praviloma želja po sodelovanju pri uresničenju razglašenih ciljev, možnost za delovanje in druženje, lahko pa so tudi rituali, simboli in karizmatične osebnosti.

Velike skupine so zanimive za nas predvsem zato, ker v njih "domujejo" tudi vsi člani malih skupin. Iz velikih skupin prinašajo člani v male skupine svoje statusne simbole, moralne vrednote in norme, včasih pa tudi točno določene vloge in pravila obnašanja. Prav zaradi teh razlik in različnih stopenj zavezanosti posameznika temu, kako se mora obnašati v malih skupinah, je delo v njih zelo težavno. Iz velikih skupin se torej prenašajo v male skupine vplivi različnih socializacijskih in resocializacijskih procesov, katerim so bili ali so še vedno izpostavljeni posamezniki, prenašata pa se tudi prevladujoča miselnost in filozofija, ki v posameznih velikih skupinah prevladujeta. Od mnogih značilnosti, ki so tipične za male skupine, je pri velikih skupinah najbolj razvit občutek pripadnosti skupini in spoštovanje njenih moralnih vrednot in norm. Med velikimi skupinami so organizacije, pa tudi večje družbene skupine, kot so sloji, narodi in družbeni razredi.

4.3 Nastanek skupin

Razlogi za ustanovitev skupin so zelo različni. Ker imamo različne tipe skupin, so tudi razlogi za njihovo ustanovitev zelo različni in niso vedno vezani na svobodno odločitev tistih, ki vanje vstopajo. Obstajajo namreč socialne skupine, v katere se posamezniki vključujejo pod določenim pritiskom ali so celo v to prisiljeni. Take skupine so v nedemokratičnih režimih tudi politične stranke, sindikalne in druge skupine za opravljanje raznih državljanskih dolžnosti kot so civilna zaščita ali služenje vojaškega roka. V te skupine ljudje ne vstopajo zaradi svojega zadovoljstva, temveč po sili zakona, ker jim sicer pretijo kazenske sankcije. Za ustanovitev skupine torej niso odločilni samo njeni cilji, temveč mnogokrat tudi aktivnosti, ki se v skupini izvajajo, in osebe, ki se v skupinah združujejo. Nekateri se vključujejo v skupine zaradi aktivnosti, ki se v njih odvijajo (kegljanje, pevska in pivska društva) ali zaradi pomembnih ljudi, ki so v skupini, da se tako lahko z njimi družijo, istovetijo in jih skušajo posnemati.

Cilji, zaradi katerih je bila skupina ustanovljena, seveda niso vedno istovetni s cilji in pričakovanji vseh članov skupine. Zato obstajajo poleg skupnih ali skupinskih ciljev tudi individualni cilji ali cilji posameznih neformalnih skupinic, ki se oblikujejo znotraj skupine. To je razlog, da obstajajo med člani skupine določene napetosti bodisi iz čisto osebnih razlogov ali zaradi različno doživljane realnosti problemov, zaradi katerih je bila skupina ustanovljena. Vsi člani skupine namreč ne doživljajo enako intenzivno pomembnosti, razsežnosti in aktualnosti problema. Zato se nekateri zelo zavzemajo za iskanje rešitev, drugi so indiferentni, tretji pa menijo,

da gre le za navidezni in ne za resnični problem, ki se ga zato ne splača reševati. Ta razhajanja mora skupina v največji možni meri odpraviti že v začetni fazi svojega konstituiranja. Za mnoge pa je skupina velika priložnost za pridobitev novega znanja in spoznanj, lažje premagovanje ovir, večjo motiviranost pri delu in za večji užitek pri doseganju zastavljenih ciljev. Če je neskladje med cilji posameznika in cilji skupine preveliko, obstaja nevarnost, da bo skupina razpadla in ne bo opravila naloge, zaradi katere je bila ustanovljena. O obstoju takih razhajanj in nezadovoljevanju svojih potreb kažejo zelo nazorno pogoste odsotnosti nekaterih članov pri delu v skupini in samovoljne izključitve nekaterih članov iz dela v skupini.

Skupine za reševanje problemov se praviloma ustanovi zato, ker je edino na ta način možno uspešno in učinkovito rešiti problem, zaradi katerega je bila skupina ustanovljena. Zapletenost in obsežnost problema je tolikšna, da ga ne more rešiti en sam človek, ker za to nima potrebnega znanja in razpoložljivih zmogljivosti. Če bi tak problem reševal posameznik, bi rešitev dobili takrat, ko morda sploh ne bi bila več potrebna, hkrati pa bi bila zaradi omejenega znanja in izkušenj premalo vsestranska in uporabna.

4.3.1 Proces izoblikovanja skupin

Z ustanovitvijo delovne skupine njeno oblikovanje ni končano. Proces oblikovanja skupine traja nekaj časa in je zelo odvisen od sestave skupine in od problemov, ki jih mora skupina rešiti. V začetku se ta proces odvija dokaj hitro, kasneje pa vse bolj počasi. Po prvih vtisih, kdo je kdo v skupini, in po okvirni seznanitvi s problemi, ki jih mora skupina rešiti, se zdi, da je skupina pripravljena za delo. Kasneje pa se izkaže, da se tedaj šele začne dejansko oblikovanje skupine.

Obstajajo razne teorije o fazah, po katerih poteka konstituiranje skupine. Morda je za ilustracijo procesov, ki se odvijajo v oblikovanju skupine, še najbolj nazorna Tuckmanova teorija [56], ki pozna 4 faze in sicer:

1. **Organizacija in informiranje.** V tej fazi se člani skupine seznanijo z nalogo, odkrijejo konkretne naloge skupine in svojo vlogo pri uresničenju teh nalog ter se seznanijo z načinom dela v skupini.

2. **Nastanek konfliktov.** V tej fazi se prične z dejanskim delom v skupini. Toda kmalu pride do medosebnih konfliktov in konfliktov, ki jih povzroči delo pri nalogi. Medosebni konflikti nastajajo med člani skupine ter tudi med člani skupine in njihovim vodjem, ker se želijo posamezniki uveljaviti bolj, kot jim je omogočeno, in ker se upirajo podrejanju nekaterim bolj uspešnim in prodornim članom

Osnovna načela za organizacijo projektnih skupin:

- Za uspeh projekta so dobri odnosi, dobro vodenje in motiviranost važnejši faktorji kot izbira razvojnih orodij in metod ter strojne opreme.
- Bolje je izbrati manjše število, toda dobrih posameznikov. To pravilo ne velja le za nogometne igralce ali obrtnike. Velike skupine niso tako produktivne kot majhne tudi zaradi potrebe po več komuniciranja in s tem tudi večjim številom možnih napak.
- Sposobnosti posameznikov je potrebno uskladiti z razpoložljivimi delovnimi nalogami. Paziti moramo, da dobri programerji na primer ne napredujejo po poklicni lestvici do položaja, ko postanejo nesposobni (Petrovo načelo). Žal je pogosto za dobre programerje edina možnost poklicnega napredovanja ta, da postanejo managerji. V interesu podjetja je, da se ponudijo možnosti poklicnega napredovanja tudi na tehničnem področju.
- Na dolgi rok se splača vlagati v ljudi, da lahko razvijejo vse svoje potenciale. Pri tem je potrebno paziti, da se posamezniki ne specializirajo preozko in da ne postanejo edini eksperti za določeno področje. Taki bodo nato prisiljeni delati samo na tem področju, kar lahko povzroči, da se naveličajo in skupino zapustijo ali pa postanejo v nekaj letih tehnično zastareli zaradi hitrega razvoja programskega inženirstva. Ljudem je potrebno omogočiti, da se stalno strokovno izpopolnjujejo.
- Paziti je potrebno, da je skupina sestavljena iz prave "mešanice" ljudi. Tako kot v nogometni ekipi, tudi v razvojni skupini ni dobro, da so sami zvezdniki.
- Tiste, ki se osebnostno ne ujamejo v skupini, je bolje izločiti.

skupine. Spori ob nalogi se pojavljajo predvsem zato, ker posamezniki menijo, da je smer iskanja rešitev neprimerna, da ni skladna z njihovimi pričakovanji in potrebami ali zato, ker je naloga zanje pretežka in ker se bojijo odgovornosti.

3. **Kohezivnost skupine.** Po pomirjenju konfliktov se porodi spoznanje, da je potrebno sprejeti člane takšne kot so in prav tako obravnavati tudi probleme, ki jih mora skupina reševati. Določena izčrpanost in utrujenost po konfliktih, ki sta prevevali skupino, se sedaj spremeni v željo po večjem sodelovanju in zavzetem iskanju rešitev. Člani se izogibajo konfliktom, nasprotovanjem in motnjam, ker želijo utrditi vlogo skupine in skupinskih norm.

4. **Končna faza razvoja grupne strukture.** Medosebni odnosi v skupini se utrjujejo, vsak sprejme svojo vlogo in si čimbolj prizadeva za uresničenje ciljev skupine. Vloge posameznikov se kombinirajo, dopolnjujejo z vlogami ostalih in tako se ustvarja v skupini harmonija. Problemi se uspešno rešujejo in iščejo se najbolj primerne rešitve. V skupini se stabilizirajo medosebni odnosi, kar je pogoj za uspešno izvajanje naloge skupine.

Odnosi v skupini se neprimerno hitreje stabilizirajo, če so bili v skupino vključeni člani s svojim pristankom, če so motivirani za delo, če nimajo posebnih zadržkov do drugih članov skupine in do naloge, ki naj jo skupina opravi. Skupina se lahko ohrani in deluje ter uresniči zastavljene naloge samo, če se pretežna večina članov v skupini zavzema za uresničitev skupnih ciljev, če razvijejo medsebojno solidarnost in če se izogibajo konfliktom. Če pa se konflikti pojavijo, jih skušajo rešiti ali čim bolj ublažiti, tako da nevtralizirajo njihove škodljive posledice. K stabilizaciji medosebnih odnosov lahko veliko prispeva tudi stabilizacija vlog in statusa posameznikov v skupini ter možnost članov skupine, da sodelujejo pri določanju nalog, ciljev in rokov. Izbira skupnih in končnih ciljev, s katerimi naj bi rešili probleme, je za vsako delovno skupino odločilnega pomena. Če imajo člani skupine veliko izkušenj in če so te izkušnje pozitivne, potem je izbira pravih ciljev neprimerno manj tvegana.

Če so posamezniki motivirani za delo v skupini in če vidijo v nalogah in ciljih možnost za svojo osebno uveljavitev, samodokazovanje, učenje in večjo varnost, bodo pripravljeni za skupino storiti kar največ. To bo zlasti opazno, ko nastopijo težave, spori, dezorientacija in druge na videz nepremagljive ovire. Tedaj bodo odkrili v sebi sposobnosti in potenciale, ki so jim bili do tedaj neznani, hkrati pa bodo storili vse, da to storijo tudi drugi člani v skupini. V takih situacijah se izjemno poveča kohezivnost v skupini, člani iščejo oporo drug v drugem, se vzajemno spodbujajo in drug na drugega pozitivno vplivajo. Mnogi primeri dokazuje, da lahko velika motiviranost prinese večje rezultate kot znanje in izkušnje nemotiviranih članov skupine.

4.3.2 Delovni pogoji

Za uspešno delovanje mora imeti skupina na voljo tudi ustrezne pogoje. Mednje sodijo predvsem prostorski pogoji, potrebna tehnična oprema ter spodbudno in tolerantno okolje. Prostorski pogoji imajo izjemen vpliv na hitro in učinkovito reševanje problemov. Če skupina deluje v prostoru, ki je izpostavljen zunanjim in notranjim motnjam (hrup, telefonski pozivi, izhodi

zaradi nujnih opravil itd.), če prostor ni dovolj velik in ustrezno opremljen, tedaj ni mogoče pričakovati velike učinkovitosti pri delu skupine.

Netolerantno in nespodbudno okolje predstavlja hudo motnjo. Če skupina deluje v okolju, ki nima potrebnega potrpljenja, ki vedno pričakuje hitre in kvalitetne rešitve, ki ne pozna tveganja, zmot in neuspelih rešitev, potem je njeno delo izpostavljeno prevelikemu in nerealnemu pritisku. Neučakanost, nepotrpežljivost pri prekoračitvi rokov, premajhna podpora, ko nastopijo težave, vse to so faktorji, ki zavirajo uspešno skupinsko delo.

4.3.3 Pomen razlik pri sestavi skupine

Kot smo že omenili, ima sestava skupine zelo velik vpliv na delo skupine. Primerno sestavljena skupina pa ni samo taka skupina, v kateri vladajo medsebojne simpatije, dopolnjevanje sposobnosti in znanja, temveč predvsem skupina, ki je ustrezno izbrana glede na problem, ki ga mora rešiti. V tem poglavju bomo proučili vpliv racionalne sestave skupine na njeno učinkovitost in sicer z vidika homogenosti ali heterogenosti skupine, starostnih razlik, razlik v znanju in izkušnjah, statusnih razlik in osebnostnih značilnosti.

Homogene in heterogene skupine

Homogene skupine poimenujemo tiste, v katerih imajo člani zelo usklajene interese, težnje in motive. Med člani v skupini ni večjih razlik glede ciljev skupine, problemov, ki jih morajo rešiti, načinov njihovega reševanja in pričakovanih rezultatov. Nasprotno pa so heterogeno sestavljene skupine tiste, kjer obstajajo med člani večje razlike tako glede ciljev kot same pomembnosti naloge in problemov, ki jih rešuje skupina in glede načina reševanja problemov.

Iz izkušenj vemo, da so heterogene skupine bolj uspešne in bolj učinkovite, predvsem pa bolj inovativne. Heterogene skupine dajejo bolj kompleksne rešitve, čeprav porabijo za delo nekoliko več časa kot homogene skupine. Pri heterogenih skupinah se problemi osvetljujejo z raznih vidikov, iščejo se rešitve v različnih smereh in rešitve, ki bodo ustrezale zahtevam vseh ali večini sodelujočih. Delo v takih skupinah je v začetku težavnejše, čas adaptacije članov na skupino in na prevladujočo miselnost je daljši in težji, toda običajno je zadovoljstvo ob zaključku naloge večje kot v homogenih skupinah.

Kadar so torej potrebne hitre, ne preveč zahtevne in čimbolj operativne odločitve, je primerneje sestaviti homogeno skupino. V ostalih primerih pa

imajo prednost heterogene skupine, zlasti za reševanje zahtevnih problemov.

Starostne razlike

Razlike v starosti članov skupine praviloma nimajo negativnega vpliva na delo v skupini, če le obstajata med člani skupine potrebna tolerantnost in strpnost. V takih skupinah se praviloma zelo uspešno kombinirata izkušenost in modrost starejših z revolucionarnostjo, zavzetostjo, znanjem in neizkušenostjo mlajše generacije. V tako sestavljenih skupinah je proces učenja zelo intenziven, od njega pa imajo vsi korist. Mlajši člani skupine lahko svoje, včasih zaletave, premalo premišljene in z zgodovinskimi izkušnjami neobremenjene rešitve ali ideje zelo uspešno kombinirajo in korigirajo z izkušnjami in znanjem starejših. Nasprotno pa starejši člani skupine ob svežih in novih idejah lažje premagujejo svojo konzervativnost in slabe izkušnje iz preteklosti.

Če v skupini ni tolerantnosti in če pride do polarizacije starejših in mlajših, je krivda praviloma na strani starejših, ki podcenjevalno gledajo na pobude mlajših zato, ker so vse preveč zaverovani v svojo nezmotljivost in modrost. V sicer splošni demokratizaciji odnosov med starejšo in mlajšo generacijo pa se pojavljajo tudi pri mlajši generaciji odpori do starejše generacije. Mlajši ocenjujejo starejše kot oviro, kot neinovativne in tiste, ki zavirajo razvoj. Nasploh pa lahko trdimo, da starostne razlike ne smejo biti ovira, ampak prej vzpodbuda za uspešno delo v skupini.

Razlike v znanju in izkušnjah

Tako kot starostne razlike tudi razlike v znanju in izkušnjah ne smejo biti ovira za delo v skupinah, če le vlada med člani skupine dovolj strpnosti in tolerantnosti. Nestrpnost in apriornost zaradi velikih razlik v znanju, ki jih posamezniki še potencirajo, predstavljajo ta za delo v skupini veliko motnjo. Velike razlike v znanju so vsaj v začetku ovira pri ustvarjanju sproščene atmosfere in pri glasnem razmišljanju. Za spremembo atmosfere, za večjo demokratičnost in sproščenost pri medsebojnem komuniciranju lahko največ storijo tisti, ki imajo največ znanja in izkušenj. Zato so za slabe odnose in nesproščeno atmosfero v skupinah največkrat krivi prav največji strokovnjaki, ker se ne znajo prilagoditi razmeram v skupini. Pri premagovanju teh razlik ima zelo pomembno vlogo vodja skupine, ki mora preprečevati potenciranje pomena tistih z večjim znanjem in izkušnjami in hkrati vzpodbujati k sodelovanju vse člane skupine. Sodelovanje v skupinah z izkušenimi in visoko strokovnimi člani je za manj izkušene člane skupine odlična priložnost,

da si pridobijo dodatno znanje in izkušnje.

Statusne razlike

Velike razlike v statusu posameznih članov skupine delujejo negativno zato, ker ustvarjajo v skupini nesproščenost, zadržanost, taktiziranje ter procese čaščenja. Status pojmujemo predvsem kot položaj, ki ga ima posameznik na hierarhični lestvici v podjetju in ne kot socialni status, ki ga ima v skupini. Za uspešno delo v skupini je priporočljivo, da so statusne razlike članov skupine čim manjše, če pa že obstajajo, potem je bolje, da so razlike posledica znanja, ne pa hierarhičnega statusa. Vključevanje šefov v delo skupine, od katerih so posamezni člani skupine posredno ali neposredno odvisni, ustvarja nesproščeno vzdušje, zadržanost in strah ter veliko taktiziranja. Četudi želijo ljudje, ki imajo višji status, to razliko čim bolj zmanjšati, jim to običajno ne uspeva, ker tisti z nižjim statusom še vedno vidijo v njih šefa oziroma osebo, od katere so sicer odvisni. Razlike v statusu praviloma negativno vplivajo na odnose v skupini.

Osebnostne značilnosti

Kot smo že omenili, za delo v skupini niso primerni vsi ljudje. Prav tako pa je bilo ugotovljeno, da nimamo zanesljivih kriterijev, s pomočjo katerih bi izbirali ljudi v posamezne skupine glede na sestavo in velikost skupine ter na problem, ki ga mora skupina reševati. Vseeno lahko trdimo, da predstavljajo v skupini veliko oviro tisti, ki ne znajo jasno in jedrnato predstaviti svojih idej, ki nimajo sposobnosti komuniciranja in vživljanja v potrebe in predstave drugih ljudi, tisti, ki so čustveno labilni, togega mišljenja, neustvarjalni, lahkoverni, polni stereotipov in ki se bojijo tveganja in odgovornosti.

4.3.4 Statusi v skupini

Kmalu po ustanovitvi skupine se oblikujejo tudi statusne strukture v skupini. O rangu posameznika v skupini obstaja zelo skladno mišljenje, ki se praviloma malo spreminja. Večina sprejema upravičenost statusnih razlik. Tisti, ki imajo visok status, si želijo ta status še bolj okrepiti, tisti, ki imajo nižji status, pa se skušajo približati ali vriniti na višji statusni nivo. Če jim to ne uspeva, si s psihološkim mehanizmom racionalizacije dokazujejo in dopovedujejo, da se to ne izplača. Kadar pa se na ta način ne morejo pomiriti ob spoznanju o svojem podrejenem položaju, nastanejo konflikti, konfrontacije in borba za višji položaj. Poleg statusa, ki ga posameznik prinese v skupino, se v skupini ta status potrdi ali okrepi glede na njegov

prispevek v skupini. Praviloma velja, da tisti, ki imajo visok startni status, skušajo ta status še bolj okrepiti in doseči, da ga skupina priznava.

Poleg objektivnega statusa obstaja še subjektivni status ali predstava posameznika o njegovem dejanskem položaju in vrednosti v skupini. Če je razlika med subjektivnim in objektivnim statusom posameznika prevelika, pride do konfliktov z ostalimi člani skupine, ker ti njegovega subjektivnega statusa ne priznavajo. Status je pomemben zato, ker tistim, ki imajo visok status, prinaša določene prednosti in privilegije. Zato se v skupini pojavljata rivalstvo in tekmovanje za status. Kadar govorimo o rivalstvu, mislimo predvsem na to, da posameznik želi pridobiti višji status, kot ga ima, ali pridobiti status, ki ga ima kdo drug, s katerim se primerja. Pri tekmovanju pa gre za to, da si posameznik želi pridobiti višji status na račun in v škodo drugega člana v skupini. Višji ali nižji status v skupini ima zelo velik vpliv tudi na konformiranje in komuniciranje. S *konformiranjem* želijo posamezniki pridobiti vsaj navidezno višji status tako, da se bolj pogosto identificirajo s tistim, ki ima višji status, poudarjajo skladnost svojih idej z njihovimi idejami, jih posnemajo v obnašanju, oblačenju in podobno. Tisti pa, ki se ne želijo ali ne morejo konformirati, delujejo nasprotno tako, da statusu oporekajo pomen in tiste, ki imajo visok status, omalovažujejo, kritizirajo in predstavljajo kot negativne osebnosti.

4.3.5 Vpliv osebnostnih značilnosti na učinkovitost dela v skupinah

Raziskave o psihosocialnem profilu oseb, ki so najbolj primerne za delo v skupinah, niso dale jasnih rezultatov. Kljub temu še vedno obstaja prepričanje, da vsakdo ni sposoben za skupinsko delo in da imajo nekateri za to delo izrazite sposobnosti in prednosti [39]. Izkušnje kažejo, da so nekatere sposobnosti, značajske značilnosti in znanja najbrž res potreben pogoj za uspešno skupinsko delo. Toda ker imamo opravka z različnimi skupinami, ki imajo različne cilje in naloge, različne načine vodenja in sestavo članov, imajo te značilnosti zelo različen pomen in vpliv na učinkovitost skupine.

Primerne osebne lastnosti so potrebne zato, ker se v skupinah poleg reševanja vsebinskih problemov rešujejo tudi medosebni odnosi, ki lahko ovirajo uspešnejše in hitrejše reševanje problemov, zaradi katerih skupina deluje. Izrazito samosvoji raziskovalci ali strokovni delavci, ki v skupinskem delu ne uživajo in vidijo v njem predvsem oviro za uresničenje svojih originalnih idej, niso primerni za skupinsko delo. Prav tako imajo velike težave s skupinskim delom tisti, ki ne znajo ali ne morejo prisluhniti idejam drugih članov in ki se niso pripravljeni neprestano učiti in spreminjati svojih spoz-

nanj ter prepričanj. Delo v skupinah zahteva inteligentne, razgledane in dinamične osebnosti, ki so pripravljene na stalni dialog, učenje in sprejemanje idej drugače mislečih.

4.4 Vodenje skupin

Potreba po vodenju v skupini je odvisna od ciljev in naloge skupine ter od njene velikosti. Obstojajo določene primarne družbene skupine neformalne narave, ki vodstva ne rabijo, ker lahko to funkcijo prevzema zdaj eden zdaj drugi član skupine. To zlasti prihaja do izraza v prijateljskih skupinah. V skupinah, ki rešujejo določene probleme, pa je vodstvo skoraj nujno, ker praksa kaže, da je na ta način skupina bolj učinkovita in hitreje rešuje zadane naloge. Seveda so tudi tu izjeme in zato tega ni mogoče posploševati.

Vodji se običajno pripisuje velik pomen za dosežke skupine. Pogosto se trdi ali misli, da imajo sposobni voditelji določene zelo tipične osebnostne in socialne lastnosti. Raziskave pa so pokazale, da take posebne značilnosti, po katerih bi lahko določali uspešne vodje, ne obstajajo. Nasprotno, pokazalo se je, da ni niti nekih skupnih osebnostnih potez niti sindroma, ki bi zagotavljal uspešno vodenje. Glavna značilnost uspešnega vodje je, da opravi vlogo, ki mu je v dani skupini dodeljena, čim bolj uspešno — da uresniči postavljene cilje in interese, ki jih skupina smatra za zelo pomembne. Da bi to funkcijo lahko opravil, mora imeti sicer določena posebna znanja, predvsem pa mora dobiti potrebno legitimno moč, ki mu zagotavlja, da bo svojo funkcijo lahko uspešno izvedel. Zato nekateri trdijo, da bi bili enako uspešni voditelji tudi drugi ljudje, če bi imeli možnost, da postanejo vodje in če bi dobili za to potrebna pooblastila.

4.4.1 Vodenje projektne skupine

Projektne skupine so sestavljene iz posameznikov, ki imajo vsi svoje osebne cilje. Da bi lahko čimbolj usklajeno delovali v projektni skupini, je važno, da vsi člani vedo, kaj se v projektni skupini od njih pričakuje. Če to ni jasno, si vsak postavi svoje cilje in kriterije uspešnosti. Ko projekt steče, se mora redno nadzorovati storilnost vseh članov skupine. Pri projektih razvoja programske skupine je napredek že tako ali tako težko meriti. Tako tudi ni dobrega recepta za ugotavljanje storilnosti posameznikov. Ob pomanjkanju boljšega merila se pogosto uporablja število programskih vrstic, napisanih v mesecu dni. Slepo upoštevanje tega merila ima lahko slabe stranske posledice, saj nastane koda, ki je daljša, kot je to res potrebno, zato pa jo je tudi težje vzdrževati in ponovno uporabljati. Boljši način

ocenjevanja in nadzorovanja ljudi je s pomočjo recenzijskih in inšpekcijskih sestankov. Te metode sicer zahtevajo več časa, vendar pa hkrati koristijo nadzoru nad celim projektom.

4.4.2 Koordinacijski mehanizmi

Organizacijske teorije [42] ločijo pet različnih organizacijskih konfiguracij, ki predstavljajo neko tipično, idealizirano okolje. Z vsako od teh konfiguracij je povezan koordinacijski mehanizem, to je želeni mehanizem za koordinacijo nalog v okviru iste organizacijske konfiguracije. Organizacijske konfiguracije in s tem povezani koordinacijski mehanizmi so:

1. **Preprosta struktura.** Skupino delavcev vodi eden ali več vodij, ki direktno nadzorujejo njihovo delo. Delavcem ni potrebno imeti niti izkušenj pri delu niti formalne izobrazbe. Za koordinacijo so odgovorni tisti, ki jih nadzorujejo. Koordinacijski mehanizem je *direktni nadzor*.

2. **Strojna birokracija.** Kadar je vsebina dela natanko določena, je možno delo organizirati s pomočjo natančnih navodil (npr. delo za tekočim trakom). Šolska izobrazba ni nujna, potrebna pa je priučitev na delovno mesto. Koordinacijski mehanizem je *standardizacija delovnega procesa*.

3. **Delitvena oblika.** Kadar je možno natančno definirati pričakovane rezultate dela, je možno delo razdeliti na več skupin (projektov), ki imajo povsem proste roke pri odločanju, kako bodo cilje dosegle. Koordinacijski mehanizem je *standardizacija delovnih rezultatov*.

4. **Profesionalna birokracija.** Kadar ni možno natančno določiti niti vsebine, niti rezultata dela, se koordinacija lahko doseže s pomočjo *standardne izobrazbe*. Določeni poklici (npr. zdravniki) imajo precejšnjo svobodo pri tem, kako izvajajo svoje delo.

5. **Pripadnost.** V velikih in inovativnih projektih je delo razdeljeno med več strokovnjakov. Vnaprej je skoraj nemogoče natančno določiti, kaj in kako naj vsak posameznik dela. Uspeh projekta je odvisen od tega, kako bo skupina kot celota dosegla vnaprej nedefinirane cilje na vnaprej nedefiniran način. Koordinacijski mehanizem je *vzajemno prilagajanje*.

Različne vrste organizacij naj bi imele različne vrste koordinacijskih mehanizmov. Seveda so zgoraj naštete konfiguracije abstraktni ideali. V resnici so lahko organizacije mešanica različnih konfiguracij.

4.4.3 Direktor in producent

Pri vodenju razvojne skupine se prepletata dve skupini nalog:

vodstvene naloge v zvezi z organiziranjem in vodenjem skupine, administriranjem in stiki z naročnikom ter

tehnične naloge v zvezi s samo naravo dela, kot je pravilna izbira metod dela in izbira pravilnih tehniških rešitev.

Obe skupini nalog sta lahko združeni v eni osebi. Pogosto pa sta ti dve skupini nalog porazdeljeni na dve osebi, saj je pri večjem projektu teh nalog preveč, težko pa je tudi najti posameznike, ki so nadarjeni za obe skupini nalog. V primeru delitve sta dve možnosti:

- Glavni vodja je zadolžen za tehnično plat. Ob sebi pa ima pomočnika, administratorja, ki skrbi za organizacijske zadeve.

- Glavni vodja je administrator, ki ima ob sebi tehničnega direktorja, ki skrbi za tehnično plat projekta. Pri tej rešitvi je pomembno, da si tehnični direktor izbori dovolj moči, da neodvisno odloča o tehničnih problemih — tako kot je pri snemanju filma važno, da lahko režiser (angl. director) dela brez prevelikega vmešavanja producenta.

4.4.4 Načini vodenja skupine

Pri vodenju skupin opazimo dva ekstremna načina in sicer demokratski in avtokratski način vodenja. Pri demokratskem vodenju je poudarek na dobrih odnosih, visoki kooperativnosti in produktivnosti skupine. Običajno so demokratsko vodene skupine bolj počasne in rabijo za svoje delo več časa. Vodja ima v takih skupinah položaj "prvega med enakimi", kar pa ne pomeni, da v izjemnih primerih, ko nastopijo težave, ne prevzame tudi nekaterih lastnosti avtokratskega vodenja.

Pri avtokratskem vodenju je osrednja in dominantna osebnost vodja skupine, ki želi vse glavne probleme rešiti sam na način, ki njemu najbolj ustreza. Od članov skupine zahteva poslušnost in pritrjevanje ter sprejemanje njegovih idej in njegovega stila vodenja brez pripomb. Skupine, ki jih vodijo avtokratski voditelji, so običajno bolj učinkovite od demokratsko vodenih, toda kvaliteta njihovih rešitev je skoraj praviloma slabša zaradi tega, ker ne izrabijo vseh potencialov, ki obstajajo v skupini. Oba primera sta dve skrajnosti, ki pa ju poznamo v praksi v raznih vmesnih izvedbah.

Tako se tudi v izrazito avtokratsko vodenih skupinah včasih pojavijo elementi demokratskega vodenja in v demokratsko vodenih skupinah prvine avtokratskega vodenja, zlasti tedaj, ko nastopijo težave.

Načine vodenja skupine lahko opredelimo tudi glede na naslednja dva parametra [52]:

1. **Medsebojni odnos**, ki označuje pozornost do posameznika in njegov odnos do drugih posameznikov v skupini.

2. **Odnos do dela.** Ta parameter pa označuje stopnjo pomembnosti cilja in načina, kako je ta cilj dosežen.

Vsak od teh dveh parametrov je lahko nizek ali visok, kar vodi do štirih osnovnih kombinacij, ki predstavljajo ekstremne načine vodenja (tab. 4.2).

Tabela 4.2: Štirje osnovni načini vodenja ljudi

		odnos do dela: nizek	odnos do dela: visok
medsebojni odnos	nizek	ločen način	požrtvovalen način
	visok	povezan način	integracijski način

Ločen način je primeren za rutinsko delo, kjer je učinkovitost prva prioriteta. Manager vodi skupino s pomočjo pravil in delovnih procedur. Ta način vodenja je primeren za koordinacijski mehanizem na osnovi standardizacije delovnega procesa.

Povezan način vodenja je učinkovit, kadar je potrebno ljudi motivirati, koordinirati njihovo delo in jih dodatno izobraževati. Delo ni rutinsko in je individualne narave. Ta način vodenja je primeren za koordinacijo na osnovi vzajemnega prilagajanja.

Požrtvovalen način vodenja je smiseln za delo v stresni situaciji. Vodja mora vedeti, kako doseči cilje, ne da bi se zameril ljudem, ki jih vodi. Primeren je za koordinacijo profesionalne birokracije.

Integracijski način vodenja je primeren, kadar rezultati dela niso določeni. Delo je razvojno ali raziskovalno in zelo prepleteno z delom drugih. Naloga vodje je, da spodbuja ljudi, kar je potrebno pri koordinaciji na osnovi vzajemnega prilagajanja.

Vsakega od teh štirih načinov vodenja lahko uporabimo pri vodenju projektov razvoja programske opreme. Za izkušeno skupino, ki razvija dobro definirano aplikacijo na področju, kjer ima že veliko izkušenj, je primerna koordinacija na osnovi standardiziranih delovnih rezultatov in ločenega načina vodenja. Za kompleksne, inovativne aplikacije, ki nastajajo pod časovnim pritiskom, pa je boljša koordinacija na osnovi vzajemnega prilagajanja in integracijskega načina vodenja.

Vodenje je zahtevna in težavna naloga, ki je ne zmore vsak. Izkušnje kažejo, da je vodenje lahko znatno lažje, če obstajajo med člani skupine in vodjem simpatije, spoštovanje in priznavanje razlik, če je naloga natančno določena, če so jasno opredeljeni njeni cilji in če naloga po svoji zahtevnosti ne presega sposobnosti ljudi, ki jo rešujejo. Znano je namreč, da je neprimerno lažje voditi operativne skupine kot pa raziskovalne in študijske skupine, ki imajo nalogo in cilje opredeljene le okvirno, način dela pa si člani skupine izbirajo sami. Lahko torej zaključimo, da vodja lažje opravlja svojo funkcijo, če ga imajo člani skupine radi, če je naloga dovolj natančno opredeljena in strukturirana in če ima vodja dovolj legalne moči in vpliva.

4.5 Odnosi in procesi v skupini

Znotraj skupin se odvijajo zelo intenzivni medosebni odnosi, katerih intenzivnost je odvisna od značaja skupine, od njenih ciljev, motiviranosti in sestave članov skupine. Posebej pomembni za skupinsko dinamiko so družbene norme, vrednote, stališča, interakcija, identifikacija in socialni pritisk.

Z normami določamo, kaj moramo storiti in česa ne smemo storiti. Norme v glavnem prepovedujejo in določajo, katera dejanja so moralno dobra in katera ne. Vsaka skupina uporablja poleg splošnih norm tudi specifične norme, ki so vezane na specifične vrednote te skupine. Z normami posamezne skupine določajo predvsem pravila obnašanja, kako mora skupina delovati kot celota in kako naj se obnašajo posamezniki glede na položaj, ki ga imajo v skupini. Norme pa so tudi pomembno merilo za ocenjevanje obnašanja vsakega posameznega člana v različnih situacijah in za presojo njegovega doprinosa k skupnim rešitvam. Skupinske norme krepijo kohezivnost skupine in povezanost članov s skupino ter solidarnost in pripravljenost, da si člani skupine vzajemno pomagajo. Splošno veljavne družbene norme, ki jih posamezniki prinašajo v skupino, so lahko tudi ovira za ustvarjanje sožitja. To se dogaja takrat, če člani skupine pripadajo različnim kulturam in če so bili izpostavljeni različnim socializacijskim procesom.

V skupini se odvija zelo močna in raznotera interakcija, s katero posamezniki skušajo delovati (vplivati) na ravnanje, obnašanje in mišljenje drugih članov v skupini. Čim bolj živahna in učinkovita je vzajemna interakcija, tem lažje je sporazumevanje in tem boljša bo socialna klima v skupini. Posebna oblika prilagoditve v skupini je *konformizem*. Značilno za konformistično obnašanje je, da tisti, ki se konformira, dela tisto, kar dela večina in se pri tem ne sprašuje, ali je to prav ali ne. Gre torej za to, da človek ravna in misli tako, kot mislijo ostali v skupini in da se pri tem ne sprašuje, kakšno je njegovo osebno mnenje ali kako bi on kot oseba moral ravnati. V zvezi s konformizmom se pojavljata še dva pojma in sicer neodvisnost in antikonformizem. Neodvisnost ali *neodvisno obnašanje* je nasprotno konformističnemu obnašanju. Značilno za neodvisno obnašanje je, da se posameznik samostojno odloča na podlagi razpoložljivih podatkov in premišljenih razlogov in ne na podlagi pritiska skupine. *Antikonformizem* pa je obnašanje, kjer posameznik zavrača vse sprejete skupinske norme, ne glede na to, ali je to upravičeno ali ne.

Čim večja je identifikacija s cilji in skupino, tem bolj učinkovito je delovanje skupine in tem boljše je počutje v skupini. Identifikacija s skupino kaže na njeno trdnost, na pripadnost članov k skupini in na to, da ima skupina zanje velik pomen. Za tiste člane, ki se ne identificirajo s skupino in nalogo, za tiste, ki ne sprejemajo ali spoštujejo ključnih moralnih norm in vrednot, uporablja skupina socialni pritisk. Oblike socialnega pritiska so zelo različne tako po vsebini kot po intenzivnosti. Od blažjih oblik pritiska, kot so opozarjanje na napake, norčevanje in večja delovna obremenitev, do bolj krutih oblik pritiska, kot so ignoriranje in izločitev iz skupine, je velik razpon. Zato se nekateri sami izločijo iz skupine, ko ugotovijo, da ne morejo prenašati socialnega pritiska in da bi njihovo vztrajanje v skupini lahko imelo negativen vpliv na njihovo delovno kariero.

4.6 Konflikti in načini njihovega reševanja

V vsaki skupini se pojavljajo občasno, v nekaterih pa tudi zelo pogosto, nasprotja, napetosti in spopadi. V skupini se ne srečujemo samo s kooperacijo in koordinacijo, temveč tudi s tekmovanjem in nasprotovanjem. Vse dotlej, dokler napetosti niso prevelike, jih skuša skupina ignorirati in obvladati z okrepljeno kohezivnostjo. Ko pa postanejo napetosti intenzivne in ko se nasprotja začnejo vidno manifestirati z medsebojnimi spopadi, tedaj govorimo o konfliktu v skupini. Konflikti nastanejo lahko med posamezniki v skupini, med skupinami in med skupino in okoljem. Kot posledica le-

Konflikti se pojavljajo v skupinah iz sledečih razlogov [56]:

- posameznik poskuša uveljaviti za druge nesprejemljive ideje,
- posameznik hoče imeti prevladujoč vpliv pri sprejemanju sklepov,
- apriorno odklanjanje predlogov in rešitev, ki jih predlaga nekdo drug,
- nepriznavanje tujih zaslug pri reševanju problemov,
- neprestano kritiziranje mišljenja skupine ali mišljenja posameznih članov skupine brez predlogov za drugačno in boljšo rešitev,
- napeta in neprijetna atmosfera v skupini.

teh pa se lahko pojavijo še medosebni konflikti, ki se manifestirajo v obliki agresivnosti do domnevnih ali resničnih povzročiteljev konfliktov. Kadar govorimo o konfliktih med skupinami, imamo v mislih predvsem odnose med skupinami, ki se ukvarjajo z istim problemom, ali pa med skupinami, ki rešujejo enake ali podobne probleme. O konfliktih znotraj skupine pa govorimo takrat, ko se med seboj spopadejo razne neformalne skupine ali posamezniki, ki se čutijo izrecno prizadeti, zaničevani ali razžaljeni.

Vsi konflikti v skupini niso negativni in rušilni. Obstajajo konstruktivni in nekonstruktivni konflikti. O nekonstruktivnih konfliktih govorimo takrat, kadar prinašajo konflikti trajne negativne posledice za posameznika ali skupino. To so konflikti, ki trajno otežujejo uspešno delovanje skupine in ogrožajo njeno delovanje ter obstoj. Tako imenovani *konstruktivni konflikti* so tisti, ki prinašajo v skupino koristi. Nedinamične skupine rabijo določene vzpodbude za povečanje svoje dinamičnosti. Te vzpodbude pa prinašajo konflikti, ki nastajajo zaradi borbe mnenj in odpora proti spremembi obstoječih pravil, ravnanj ali rešitev. Konflikti torej omogočajo razvoj organizacije in napredek. Značilno je, da ljudje včasih upravičeno zahtevajo in ustvarjajo situacije, v katerih vidijo možnost za odpravo obstoječe, zastarele in času neprimerne miselnosti ali rešitev. Konflikti tudi omogočajo, da se spremenijo neustrezne vloge v skupini ali moralne vrednote.

V skupini se pojavljajo tudi medosebni konflikti, ki nimajo značaja skupinskih konfliktov. Ti običajno niso vezani na cilje in naloge skupine ali položaje in formalne odnose posameznikov, ki so v konfliktu. To so spori, ki nastajajo zaradi osebne nestrpnosti, neprilagodljivosti med ljudmi, ki delajo v skupini, ki se neprestano srečujejo in ki trpijo zaradi teh spopadov. Medosebni konflikti imajo velik negativni vpliv na skupino in atmosfero v skupini zlasti tedaj, če so to konflikti med osebami, ki imajo visok položaj. Zelo negativen vpliv na odnose v skupini imajo konflikti, ki nastanejo znotraj

Glavni razlogi za medosebne konflikte so naslednji [56]:

- borba med posamezniki ali skupinami zaradi denarja, moči, posesti in prostora, zato ker ne obstaja pripravljenost za delitev in želijo nekateri vse zadržati zase;
- različno pojmovanje vsebine posameznih nalog zaradi odpora, ki ga čuti del skupine, zaradi različnih okusov in pričakovanj, ki niso sprejemljivi za vse člane skupine;
- različno pojmovanje vrednot, zato ker ne obstaja soglasje o temeljnih vrednotah, ki jih morajo vsi spoštovati;
- različno prepričanje in ocene, kaj je pomembno, kaj je nepomembno, kaj je pravilno ali nepravilno;
- razlike v položajih posameznika ali skupine v organizaciji, pri čemer gre predvsem za borbo za večjo oblast in vpliv.

skupine zaradi želje po večjem vplivu, prestižu ali večji priljubljenosti.

V kolikšni meri je možno rešiti konflikt, zavisi od mnogih dejstev. Najpomembnejša med njimi so značaj konflikta, njegova intenzivnost, izgledi za možnost sporazuma in vzajemno zaupanje, osebne značilnosti oseb, ki so v konfliktu in drugo. Če imata obe strani, ki sta v konfliktu, zelo visoke težnje in aspiracije, potem bo težje prišlo do sporazuma, kot če imata obe strani nižje aspiracije ali če ima ena od njiju znatno nižje zahteve. Sporazumevanje je tudi težje, če rešujeta spor osebi, ki sta zelo agresivni, avtoritarni, če imata veliko potrebo po dominaciji, po dogmatizmu ali če sta zelo nezaupljivi. Nasprotno pa je sporazumevanje neprimerno lažje, če gre za osebi, ki zaupata druga drugi, ki sta odprti za pobude tistih, ki težijo k enakosti, pravičnosti in imajo pozitiven odnos do sogovornika.

Včasih se na konflikte odzivajo člani skupine tudi tako, da ustvarjajo koalicije. Koalicije same po sebi ne pomenijo reševanja konflikta, temveč so znak njegovega zaostrovanja. Pod koalicijo razumemo združevanje dveh ali več članov oziroma dveh ali več podskupin v okviru neke skupine. Namen koalicije je, da bi združeni dosegli cilje, ki jih brez združitve ne bi mogli doseči, zato ker se bistveno razlikujejo od ciljev ostalih članov skupine. Da lahko govorimo o koaliciji, morajo obstajati najmanj tri osebe ali tri podskupine z različnimi pojmovanji o tem, kako rešiti nek problem ali kako ravnati v določeni situaciji. Najmanj dve osebi ali dve podskupini se združita v koalicijo proti ostalim, da bi jima njuna združena akcija prinesla korist, ki jo sicer nobena od njiju ne bi mogla doseči. Koalicije se običajno pojavljajo v večjih skupinah, niso pa redek primer tudi v manjših skupinah. Koalicije so

Za reševanje konfliktov obstaja več preizkušenih metod, najbolj pogoste in uspešne metode so naslednje [56, 14]:

- Sporno zadevo se skuša rešiti z *glasovanjem* tako, da je sprejeta tista rešitev, za katero se je odločila večina. Ta način ni vedno najboljši, čeprav se situacijo da odločiti z glasovanjem. Glasovanje namreč ni vedno odraz dejanskih stališč, ker se dostikrat izvaja glasovanje pod pritiskom, pod vplivom najbolj vplivnih oseb in zaradi potrebe po konformizmu. Glasovanja je neprimerno tudi zato, ker po izidu glasovanja tisti, ki so bili "poraženi", običajno ne spremenijo svojega mišljenja in svojih teženj. Glasovanje je torej koristno samo takrat, ko pomeni poenotenje obnašanja in ravnanja in če skupina spozna, da je takšno stanje boljše od stanja, ki ga povzroča konflikt.
- Zelo učinkoviti in koristni so *pogovori* med stranema, ki sta v konfliktu. Vsaka od prizadetih strani skuša v razgovoru najti argumente, s katerimi skuša prepričati drugo stran ali pa popusti v znak dobre volje v nekaterih svojih prvotnih zahtevah.
- Ena od strani, ki so v sporu, *odstopi od nekaterih svojih prvotnih zahtev*, ne da bi zahtevala, da to stori tudi druga stran. Tak način izkazovanja dobre volje brez pogojev običajno vzbudi pri nasprotni strani željo po posnemanju in tako se postopoma spor zglajuje. Če skuša to pripravljenost druga stran izkoristiti za to, da zadrži prednost, se običajno tak poskus reševanja konflikta prekine.
- Obe strani skušata najti v novem skupnem cilju uresničenje *obojestranskih koristi*. Najdba takega skupnega cilja "višjega ranga ", ki bi zadovoljil obe strani, je zelo težka naloga, toda konflikt je potem sorazmerno hitro rešljiv.

znak, da se je konflikt zaostril in da je to po mnenju prizadetih strani najbolj primeren način za reševanje konflikta v skupini. Koalicije lahko pomenijo tudi konec sožitja v skupini in razpad skupine.

Poglavje 5

Mrežno načrtovanje

Mrežno načrtovanje je analitično orodje za načrtovanje, spremljanje in kontrolo projektov. Cilj mrežnega načrtovanja je racionalna uporaba zmogljivosti, časa in stroškov projekta. Mrežni načrt, ki pri tem nastane, pa omogoča jasen pregled strukture projekta. Izdelava mrežnega načrta spada v fazo definiranja projekta, ko se izdela izvedbeni ali osnovni načrt projekta. Mrežni načrt je tako bistveni del načrtovanja vsakega zahtevnega projekta. Med izvajanjem projekta pa mrežni načrt omogoča v vsakem trenutku vpogled v stanje projekta, kar je nujno za nadzor projekta, ko se je potrebno odločiti tudi o morebitnih spremembah, da bi vseeno lahko dosegli zastavljene cilje [67, 34, 21, 36].

Razvoj mrežnega planiranja je povezan z začetki formalnega študija projektnega dela koncem 50-tih let tega stoletja. Dve metodi mrežnega planiranja, ki sta bili razviti takrat in sta še vedno osnova vsem sodobnim računalniškim programom za načrtovanje projektov, sta metodi CPM in PERT. Osnovni koraki so pri obeh metodah mrežnega planiranje enaki in jih bomo po vrsti obravnavali v nadaljevanju tega poglavja. Sedaj omenimo le osnovne značilnosti in razlike med tema dvema klasičnima metodama mrežnega planiranja.

Metoda CPM

Critical Path Method (CPM) ali *metoda kritične poti* je bila razvita leta 1957 v ZDA za kontrolo pri izgradnji in vzdrževanju velikih kemičnih tovarn[1]. CPM je deterministična metoda, uporabna predvsem tam, kjer je možno natančno oceniti čas trajanja posameznih aktivnosti.

[1] Podjetje DuPont & Rand

Metoda PERT

Program Evolution and Review Technique (PERT) ali *metoda ocene in revizije programa* je bila razvita leta 1958 za potrebe ameriške vojne mornarice, kjer je bilo potrebno koordinirati delo stotine izvajalcev. PERT je stohastična metoda, saj ni potrebno natančno določiti časa trajanja posamezne aktivnosti. Za vsako aktivnost v mrežnem načrtu PERT moramo določiti **optimističen** čas (10% verjetnost, da se izpolni), **najbolj verjeten** čas in **pesimističen** čas (zopet 10% verjetnost, da se izpolni). Če privzamemo običajno predpostavko metode PERT, da imajo časi trajanja aktivnosti distribucijo β, lahko izračunamo pričakovani čas t_e za vsako aktivnost po formuli

$$t_e = \frac{t_o + 4t_v + t_p}{6}, \tag{5.1}$$

oziroma varianco trajanja aktivnosti

$$v = \left(\frac{t_o + 4t_v + t_p}{6}\right)^2, \tag{5.2}$$

kjer so

t_o – optimistični čas za dokončanje aktivnosti,
t_v – najbolj verjetni čas za dokončanje aktivnosti,
t_p – pesimistični čas za dokončanje aktivnosti,
t_e – pričakovani čas za dokončanje aktivnosti,
v – varianca časa za dokončanje aktivnosti.

Potek mrežnega načrtovanja

Mrežno načrtovanje poteka po naslednjih štirih korakih:

1. **Analiza strukture projekta.** Cilj te analize je narediti mrežni diagram z vsemi aktivnostmi, potrebnimi za dosego ciljev projekta. Struktura mrežnega diagrama ponazarja povezanost med aktivnostmi.

2. **Časovna analiza.** V mrežni diagram vnesemo ocene trajanja posameznih aktivnosti. Cilj časovne analize je odkriti kritično pot skozi mrežni diagram, to je zaporedje vseh tistih aktivnosti, ki določajo trajanje celotnega projekta.

3. **Analiza zmogljivosti.** V mrežni diagram vnesemo še ocene potrebnih zmogljivosti za vsako posamezno aktivnost. Cilj analize zmogljivosti je odkriti, kakšne so potrebe po zmogljivostih celotnega projekta v posameznih časovnih obdobjih. Običajno želimo zmogljivosti obremeniti čimbolj enakomerno ves čas trajanja projekta. Po potrebi moramo mrežni diagram uskladiti z razpoložljivimi zmogljivostmi ali določiti, koliko dodatnih zmogljivosti potrebujemo, če želimo projekt dokončati v najkrajšem možnem času.

4. **Analiza stroškov.** V mrežni diagram vnesemo na koncu še stroške oziroma cene zmogljivosti na časovno enoto in morebitne z aktivnostmi povezane fiksne stroške. Mrežni diagram optimiziramo še glede na pritok in odtok sredstev, tako da so stroški financiranja projekta čim manjši.

V nadaljevanju si oglejmo podrobno štiri korake v poteku mrežnega načrtovanja.

5.1 Analiza strukture projekta

Analiza strukture projekta zajema:

1. razčlenitev projekta glede na cilje in podcilje projekta in določitev aktivnosti;

2. ugotovitev kako so aktivnosti med seboj povezane;

3. izdelavo mrežnega načrta.

5.1.1 Določitev aktivnosti

Kako določimo aktivnosti, zavisi predvsem od narave projekta. Aktivnosti naj bodo čim bolj zaokrožene delovne naloge z jasnimi in merljivimi cilji, za katere potrebujemo določeno vrsto zmogljivosti, časa in sredstev. Dva osnovna načina razdelitve projekta na aktivnosti sta funkcijsko in objektno strukturiranje.

Funkcijsko strukturiranje

Funkcijsko strukturiranje dela se izvaja glede na vrsto dela, naprimer: projektiranje, priprava, nabava, izdelava, montaža. Pri razvoju programske opreme so to lahko analiza, načrtovanje, kodiranje in testiranje.

Aktivnosti naj bodo definirane tako, da je možno:

- določiti izvajalca in odgovorne osebe,
- določiti raven vodenja izvajalskih enot,
- načrtovati obremenitve vseh izvajalskih enot oziroma vseh zmogljivosti,
- načrtovati stroške tako, do bo možno enolično vršiti obračun po aktivnostih,
- določiti čas trajanja,
- enolično ugotoviti rezultate,
- določiti medsebojno odvisnost aktivnosti,
- določiti verjetnost realizacije aktivnosti in dosego ciljev projekta,
- določiti potrebne zmogljivosti.

Objektno strukturiranje

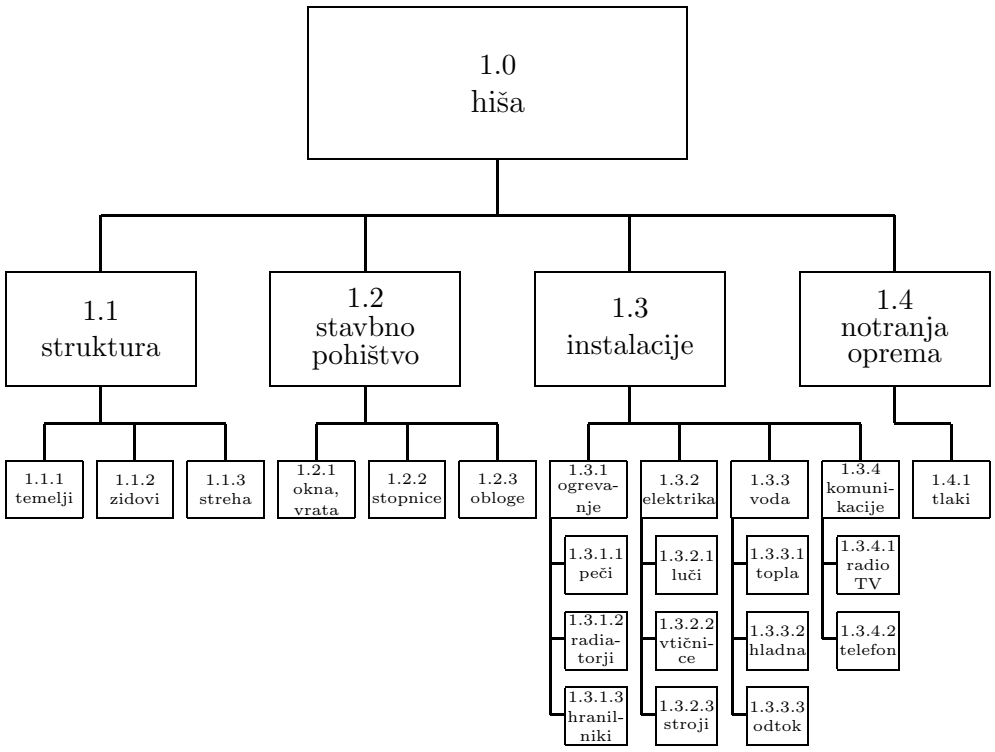
Objektno strukturiranje se izvaja glede na objekt dela. Projekt se naprimer razdeli na podprojekte, ki predstavljajo sestavne dele, posamezne objekte ali sklope. V primeru razvoja programske opreme so to lahko uporabniški vmesnik, podatkovne zbirke, komunikacijski moduli itd.

Razširjena objektno zasnovana metoda za definiranje potrebnega dela je metoda WBS (*Work Breakdown Structure*). Cilji projekta v obliki konkretnih izdelkov ali storitev so postavljeni na vrh drevesnega diagrama. Vsak segment nato postopoma delimo na ožje podcilje toliko časa, dokler ne pridemo do skupin opravil, ki tvorijo naravno (funkcijsko) zaključene delovne celote. Za vsako od teh naravnih delovnih celot nato definiramo potrebno aktivnost. Za označevanje segmentov v drevesni strukturi ponavadi uporabljamo hierarhičen način označevanja oziroma številčenja. Po drevesni strukturi lahko zlahka seštevamo zmogljivosti za segmente na različnih nivojih hierarhije. Primer diagrama WBS je na sliki 5.1.

5.1.2 Povezanost aktivnosti

Povezanost med aktivnostmi (kako si sledijo, ali lahko tečejo vzporedno) dobimo lahko na dva načina:

- s tehniko vprašanj ali
- z matrično tehniko.



Slika 5.1: Delovna struktura—WBS (Work Breakdown Structure) za gradnjo hiše

Tehnika vprašanj

Da bi odkrili medsebojno soodvisnost aktivnosti s tehniko vprašanj, moramo za vsako aktivnost odgovoriti na naslednjih pet vprašanj:

1. Ali je opazovana aktivnost **začetna** aktivnost?

2. Ali je opazovana aktivnost **zaključna** aktivnost?

3. Katere aktivnosti se morajo **končati** neposredno pred opazovano aktivnostjo?

4. Katere aktivnosti se morajo **pričeti** tik po zaključku opazovane aktivnosti?

5. Katere aktivnosti lahko potekajo **vzporedno** opazovani aktivnosti?

Rezultate analize medsebojne odvisnosti aktivnosti zapisujemo v tabelo (tabela 5.1) ali pa kar sproti konstruiramo mrežni načrt.

Matrična tehnika

Pri odkrivanju medsebojne soodvisnosti aktivnosti z matrično metodo je potrebno ugotoviti, katere aktivnosti morajo biti **dokončane**, preden se začne opazovana aktivnost. Ugotovitve beležimo v prednostno matriko. Primer take matrike je v tabeli 5.2. Znak $\times$ v matriki pove, da mora biti aktivnost v vrstici končana, preden se prične odgovarjajoča aktivnost v stolpcu. V prednostni matriki smejo biti oznake $\times$ le nad glavno diagonalo matrike. Če pri reševanju dobimo oznake $\times$ tudi pod glavno matriko, jo moramo urediti z ustrezno zamenjavo vrstic in stolpcev. Na osnovi urejene prednostne matrike je možno sestaviti ustrezni mrežni diagram.

5.1.3 Dogodkovna in aktivnostna mreža

Povezanost aktivnosti v projektu grafično prikažemo z aciklično usmerjenim grafom. Možna sta dva načina prikaza: z dogodkovnimi ali z aktivnostnimi mrežnimi diagrami.

V **dogodkovnih mrežah** so dogodki postavljeni v vozlišča grafa, *aktivnosti* pa so prikazane kot *povezave* med posameznimi dogodki (slika 5.2). Dogodkovne mreže so včasih označene z angleško kratico AOA *(Activity On Arrow diagram)* ali kar *arrow diagram* [20]. V tabelah in mrežnih diagramih označujemo aktivnosti z velikimi tiskanimi črkami, dogodke pa z naraščajočimi številkami. Pri oštevilčenju ponavadi uporabljamo mnogokratnike števila 10, tako da kasneje lažje dopolnjujemo ali spreminjamo diagrame.

V **aktivnostni mreži** so *aktivnosti* postavljene *v vozlišča* grafa (slika 5.3), povezave pa določajo vrstni red aktivnosti. Aktivnostne mreže poznamo tudi pod kratico AON *(Activity On Node diagrams).*

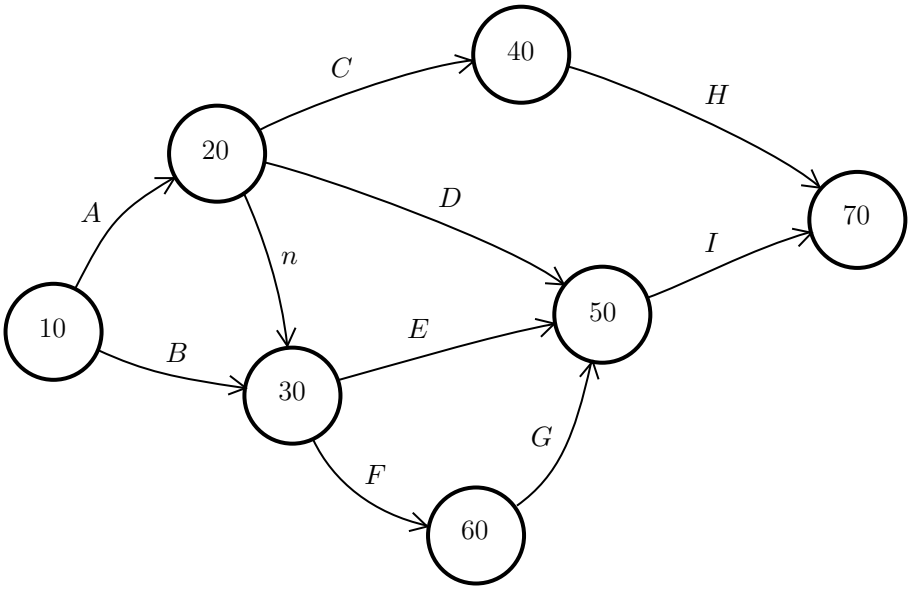
Obe vrsti mrež se v bistvu ne razlikujeta. Eno vrsto mreže lahko prevedemo v drugo in obratno. Čeprav so se dogodkovne mreže spočetka bolj uporabljale, pa so danes aktivnostne mreže bolj razširjene, še posebej ker so bolj primerne za uporabo v računalniških programih. V dogodkovnih mrežah so včasih potrebne tako imenovane *navidezne aktivnosti*, ki imajo čas trajanja enak 0, da bi lahko pravilno prikazali soodvisnot vseh aktivnosti. Tako smo na primer v dogodkovno mrežo na sliki 5.2 morali uvesti navidezno

Tabela 5.1: Rezultat analize odvisnosti med aktivnostmi se lahko zapiše v obliki tabele. V tabeli so označeni še časi trajanja aktivnosti in potrebne zmogljivosti za izvajanje aktivnost, ki jih bomo kasneje.

Opazovana aktivnost	Neposredne predhodnice	Trajanje aktivnosti	Potrebne zmogljivosti
A	–	2	9
B	–	3	5
C	A	3	3
D	A	5	4
E	A, B	2	6
F	A, B	1	10
G	F	4	7
H	C	3	5
I	D, E, G	2	9

Tabela 5.2: Prednostna ali precedenčna matrika (gre za isti projekt kot v tabeli 5.1) pove, katere aktivnosti se morajo dokončati pred obravnavano aktivnostjo. Znak $\times$ v matriki pove, da mora biti aktivnost v vrstici končana, preden se prične aktivnost v stolpcu.

Po zaključku aktivnosti:	se pričnejo naslednje aktivnosti:								
	A	B	C	D	E	F	G	H	I
A	•		$\times$	$\times$	$\times$	$\times$			
B		•			$\times$	$\times$			
C			•					$\times$	
D				•					$\times$
E					•				$\times$
F						•	$\times$		
G							•		$\times$
H								•	
I									•

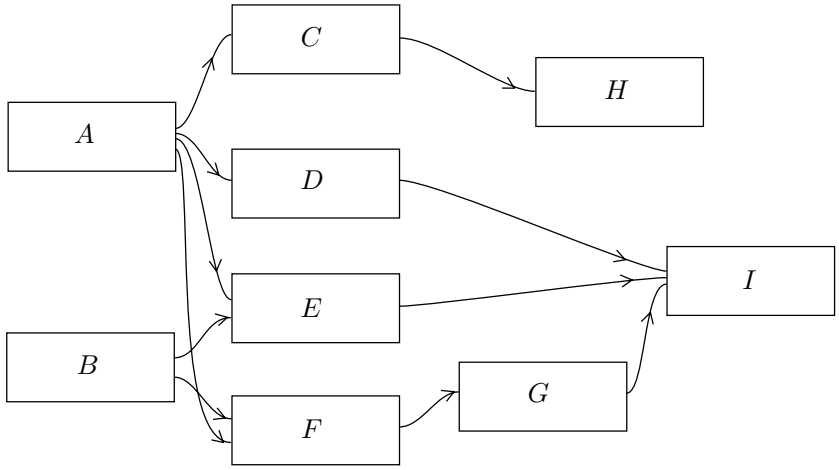


Dogodkovna mreža je definirana z naslednjimi pojmi:

dogodki:	$10, 20, 30 \ldots$
aktivnosti:	$A, B, C \ldots$
	čas trajanja aktivnosti $t_{ij} \geq 0$,
	prirejene zmogljivosti $r_{ij} \geq 0$,
	stroški aktivnosti $s_{ij} \geq 0$,
	kjer i in j definirajo aktivnosti
	med ustreznimi dogodki $10, 20, 30 \ldots$
navidezna aktivnost:	n, kjer velja: $t_{ij} = 0, r_{ij} = 0, s_{ij} = 0$

Slika 5.2: Dogodkovna mreža za aktivnosti iz tabel 5.1 in 5.2.

aktivnost 20-30, saj je začetek aktivnosti C in D odvisen le od konca aktivnosti A, začetek aktivnosti E in F pa je odvisen od konca aktivnosti A **in** B! Navidezna aktivnost 20-30 pa postane nepotrebna, ko problem predstavimo z aktivnostno mrežo (slika 5.3).



Aktivnostna mreža je definirana z naslednjimi pojmi:

aktivnosti: $A, B, C \ldots$
čas trajanja aktivnosti $t_X \geq 0$,
prirejene zmogljivosti $r_X \geq 0$,
stroški aktivnosti $s_X \geq 0$,
in aktivnost $X \in \{A, B, C \ldots\}$

Slika 5.3: Aktivnostna mreža za aktivnosti iz tabele 5.1, ki je enakovredna dogodkovni mreži na sliki 5.2.

Primerjava dogodkovne in aktivnostne mreže:

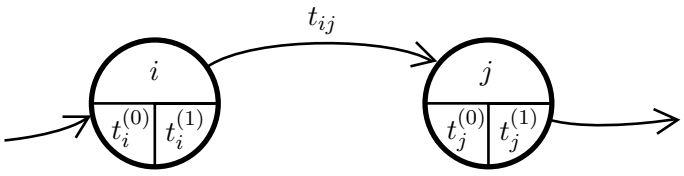
- Strukturo aktivnostne mreže je lažje razumeti.
- V povprečju potrebuje aktivnostna mreža manj elementov kot dogodkovna mreža. Namesto 100 aktivnosti in 150 dogodkov v dogodkovni mreži je v aktivnostni mreži v povprečju potrebno zgolj okoli 90 aktivnosti .
- Navidezne aktivnosti v aktivnostni mreži niso potrebne.
- Logične napake se v aktivnostni mreži lažje odkrijejo.
- Vnos podatkov v računalnik je enostavnejši pri aktivnostni mreži.
- Spremljanje izvajanja projekta je pri aktivnostni mreži lažje, saj ugotavljamo le spremembe aktivnosti, ne pa tudi dogodkov.

5.2 Časovna analiza

V mrežnem diagramu smo do sedaj predstavili le strukturo projekta, to je medsebojno odvisnost vseh aktivnosti. Da bi izvedeli čas trajanja celotnega projekta in ugotovili, kdaj se morajo posamezne aktivnosti začeti izvajati, pa moramo vedeti čas trajanja vsake aktivnosti. Natančno ocenjevanje trajanja aktivnosti je težko in včasih celo nemogoče. Kako natančno in zanesljivo ocenjujemo trajanje aktivnosti, zavisi v veliki meri od vrste aktivnosti in vrste projekta. Veliko vlogo pri ocenjevanju igrajo izkušnje, pridobljene pri prejšnjih podobnih projektih. Na srečo zelo velika natančnost časovnih ocen ni nujna za uspešno mrežno planiranje. Metoda PERT, o kateri smo govorili na strani 70, uporablja za to tri ocene za čas trajanja aktivnosti: pesimistično, verjetno in optimistično. Te ocene so osnova za izračun pričakovanega časa in časovne variance vsake aktivnosti.

Časovna analiza dogodkovnih in aktivnostnih mrež se nekoliko razlikuje, saj pri časovni analizi dogodkovne mreže dobimo najzgodnejše oziroma najkasnejše roke nastopanja posameznih dogodkov in moramo najzgodnejše oziroma najkasnejše roke začetka in zaključka aktivnosti šele izračunati. Časovna analiza aktivnostne mreže pa nam da te rezultate neposredno.

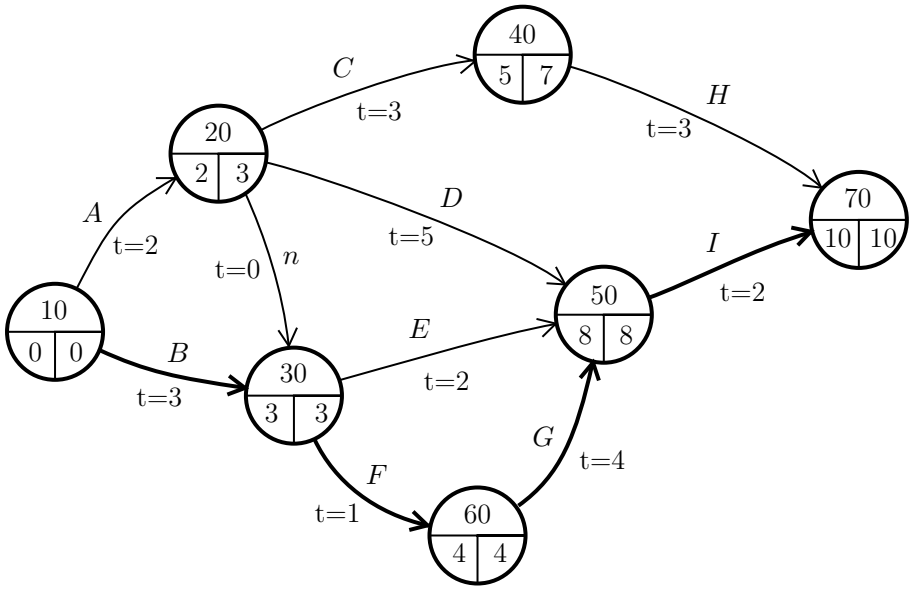
5.2.1 Časovna analiza dogodkovnih mrež



Slika 5.4: Element dogodkovne mreže

Elementi dogodkovne mreže so prikazani na sliki 5.4, kjer oznake pomenijo:

i	začetni dogodek aktivnosti *i-j*,
j	zaključni dogodek aktivnosti *i-j*,
t_{ij}	čas trajanja aktivnosti *i-j*,
$t_i^{(0)}$	najzgodnejši rok nastopanja dogodka i,
$t_j^{(0)}$	najzgodnejši rok nastopanja dogodka j,
$t_i^{(1)}$	najkasnejši rok nastopanja dogodka i,
$t_j^{(1)}$	najkasnejši rok nastopanja dogodka j.



Slika 5.5: Primer analize kritične poti v dogodkovnem mrežnem diagramu za projekt, ki je definiran z aktivnostmi v tabeli 5.1. Kritična pot je označena z odebeljenimi puščicami.

Za prvi dogodek velja, da je $t_1^{(0)} = 0$, za zadnji n-ti dogodek pa $t_n^{(0)} = t_n^{(1)}$.

Po mreži naprej za določanje časa trajanja celotnega projekta

Da bi izračunali čas trajanja celotnega projekta in najzgodnejše roke nastopanja dogodkov začnemo pri začetnem dogodku in seštevamo čase trajanja aktivnosti po mreži do konca projekta.

Najzgodnejše roke nastopanja dogodkov določamo *progresivno* po enačbi:

$$t_j^{(0)} = \max\{t_i^{(0)} + t_{ij}\} \tag{5.3}$$

pri tem pa je

$i = 1, 2, \ldots, n-1$
$j = 2, 3, \ldots, n$ in $i \neq j$
n je število dogodkov v mreži.

Enačba 5.3 pomeni, da se pri računanju najzgodnejših rokov nastopanja dogodkov pomikamo po mreži naprej in na vsaki povezavi prištejemo čas trajanja dotične aktivnosti. Če v neko vozlišče vodi več poti, potem v tem

vozlišču upoštevamo največjo vsoto. Rezultat te analize po mreži naprej je izračun *najkrajšega možnega časa trajanja* celotnega projekta.

Po mreži nazaj za določitev kritične poti

V mrežnem diagramu niso vse aktivnosti enako pomembne. Nekatere lahko zamujajo, ne da bi vplivale na najkrajši možni čas dokončanja projekta. Toda v vsakem mrežnem diagramu obstaja zaporedje aktivnosti, ki neposredno določajo trajanje celotnega projekta. Če zamuja katera od kritičnih aktivnosti, se zavleče celoten projekt. Tej poti skozi mrežni diagram pravimo **kritična pot**. Kritično pot določimo tako, da mrežni diagram sedaj "prečešemo" še od zadaj naprej.

Najkasnejše roke nastopanja dogodkov določamo *retrogradno* po enačbi:

$$t_i^{(1)} = \min\{t_j^{(1)} - t_{ij}\} \tag{5.4}$$

pri tem pa je

$$\begin{aligned} &t_n^{(1)} = t_n^{(0)} \\ &i = n-1, n-2, \ldots, 1 \\ &j = n, n-1, \ldots, 2 \text{ in } i \neq j. \end{aligned}$$

Direktna implementacija enačb 5.3 in 5.4 vodi v rekurzivni algoritem z eksponentno časovno zahtevnostjo v odvisnosti od velikosti projekta. Z dinamičnim programiranjem lahko izračunamo dolžino časa projekta in kritično pot v linearnem času [32]. Enačba 5.4 pomeni, da se v diagramu vračamo po vseh možnih poteh od konca proti začetku projekta, tako da v vozliščih odštevamo čase prehojenih aktivnosti. Kjer je zaradi večih poti v vozlišče možno več različnih rezultatov, upoštevamo v tem vozlišču najmanjšo razliko. Kritična aktivnost je tista, za katero velja da je

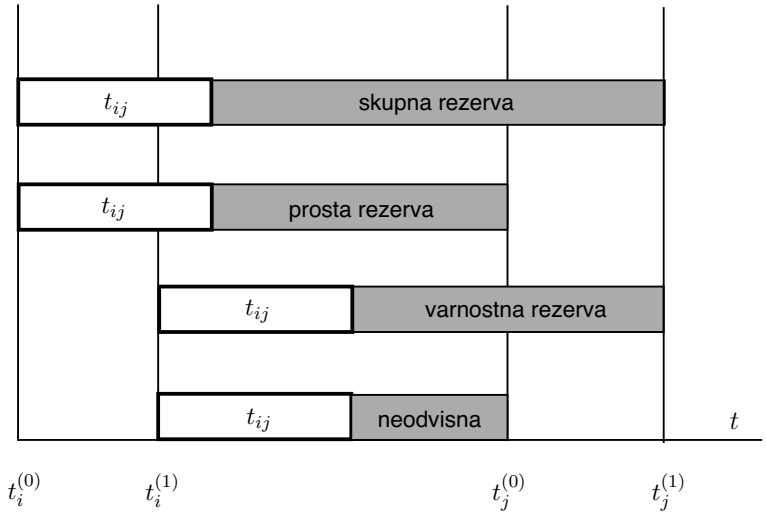
$$t_i^{(0)} = t_i^{(1)} \tag{5.5}$$

$$t_j^{(0)} = t_j^{(1)} \tag{5.6}$$

$$t_j^{(1)} = t_i^{(0)} + t_{ij}. \tag{5.7}$$

Kritična pot je najdaljša pot v mrežnem diagramu, ki vsebuje same kritične aktivnosti.

Primer časovne analize projekta, ki je definiran v tabeli 5.1, s pomočjo dogodkovne mreže je na sliki 5.5.



Slika 5.6: Štiri vrste časovnih rezerv v dogodkovni mreži

Časovne rezerve v dogodkovnem diagramu

Iz najzgodnejših in najkasnejših rokov začetkov in koncev dogodkov lahko izračunamo štiri vrste časovnih rezerv:

skupna rezerva aktivnosti *i-j* $\quad T_{td} = t_j^{(1)} - t_i^{(0)} - t_{ij} \geq 0,$

prosta rezerva aktivnosti *i-j* $\quad T_{pd} = t_j^{(0)} - t_i^{(0)} - t_{ij} \geq 0,$

varnostna rezerva aktivnosti *i-j* $\quad T_{vd} = t_j^{(1)} - t_i^{(1)} - t_{ij} \geq 0,$

neodvisna rezerva aktivnosti *i-j* $\quad T_{nd} = t_j^{(0)} - t_i^{(1)} - t_{ij} \geq 0,$

Te štiri vrste časovnih rezerv si najpreprosteje zapomnimo oziroma razložimo tako, da je *totalno* drsenje aktivnosti odvisno tako od *predhodnih* aktivnosti kot od aktivnosti, ki neposredno *sledijo*. *Prosto* drsenje je odvisno le od aktivnosti, ki neposredno *sledijo*. *Varnostno* drsenje pa je odvisno le od *predhodnih* aktivnosti. *Neodvisno* drsenje pa *ni odvisno* niti od predhodnih aktivnosti niti od aktivnosti, ki sledijo. Kritične aktivnosti nimajo nobene časovne rezerve! Grafična ponazoritev časovnih rezerv v dogodkovni mreži je na sliki 5.6.

Iz primera projekta, definiranega s tabelo 5.1, izberimo aktivnost C oziroma aktivnost 20-40 in izračunajmo njene časovne rezerve:

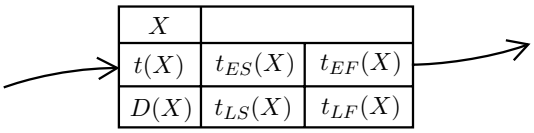
skupna: $T_{td} = t_{40}^{(1)} - t_{20}^{(0)} - t_{ij} = 7 - 2 - 3 = 2,$

prosta: $T_{pd} = t_{40}^{(0)} - t_{20}^{(0)} - t_{ij} = 5 - 2 - 3 = 0,$

varnostna: $T_{vd} = t_{40}^{(1)} - t_{20}^{(1)} - t_{ij} = 7 - 3 - 3 = 1,$

neodvisna: $T_{nd} = t_{40}^{(0)} - t_{20}^{(1)} - t_{ij} = 5 - 3 - 3 = -1$ oz. pod defin. 0.

5.2.2 Časovna analiza aktivnostnih mrež



Slika 5.7: Element aktivnostne mreže

Element aktivnostne mreže je prikazan na sliki 5.7, kjer pomeni:

X oznako aktivnosti,

$t(X)$ čas trajanja aktivnosti X,

$t_{ES}(X)$ najzgodnejši rok začetka aktivnosti X, [2]

$t_{EF}(X)$ najzgodnejši rok dokončanja aktivnosti X, [3]

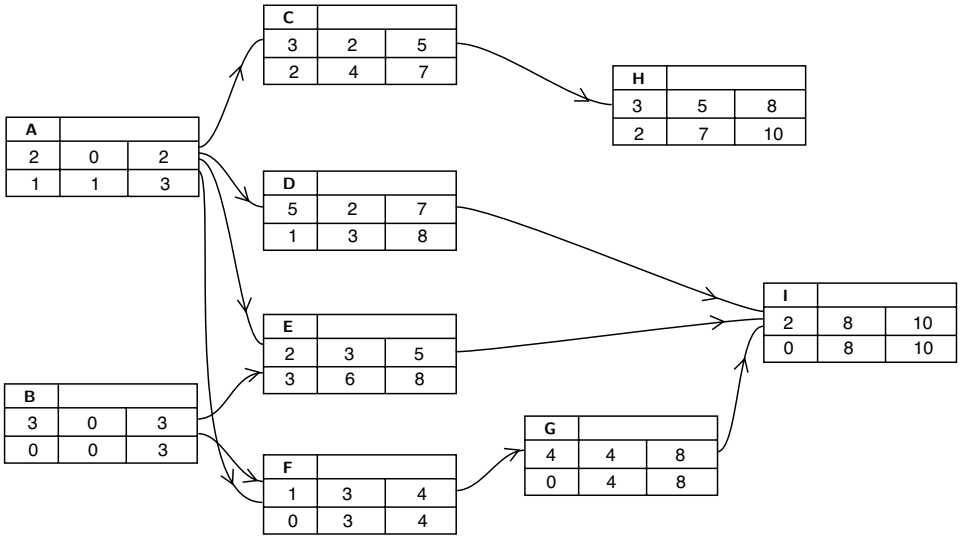
$t_{LS}(X)$ najkasnejši rok začetka aktivnosti X, [4]

$t_{LF}(X)$ najkasnejši rok dokončanja aktivnosti X, [5]

$D(X)$ drsenje aktivnosti X.

Določanje najzgodnejših rokov aktivnosti poteka *progresivno* (naprej po mrežnem diagramu) po naslednjem zaporedju:

[2] ES – Earliest Start
[3] EF – Earliest Finish
[4] LS – Latest Start
[5] LF – Latest Finish



Slika 5.8: Analiza kritične poti v aktivnostnem mrežnem diagramu za projekt, definiran z razpredelnico v tabeli 5.1.

1. Prvi aktivnosti A pripišemo najzgodnejši rok začetka aktivnosti

$$t_{ES}(A) = 0 \tag{5.8}$$

   in izračunamo najzgodnejši rok konca aktivnosti

$$t_{EF}(A) = t_{ES}(A) + t(A) \tag{5.9}$$

2. Najzgodnejši rok začetka ostalih aktivnosti določimo po naslednji enačbi:

$$t_{ES}(V) = \max_{U \in \mathcal{P}}(t_{ES}(U) + t(U)) \tag{5.10}$$

   kjer je $\mathcal{P}$ množica vseh neposrednih predhodnikov aktivnosti V.

3. Najzgodnejši rok zaključka aktivnosti določimo na osnovi najzgodnejših rokov nastopanja teh aktivnosti

$$t_{EF}(X) = t_{ES}(X) + t(X) \tag{5.11}$$

Najkasnejše roke nastopanja aktivnosti pa določimo zopet *retrogradno* in sicer:

1. Za zadnjo aktivnost Z velja

$$t_{LF}(Z) = t_{EF}(Z) = t_{projekta} \tag{5.12}$$

2. Nato določimo najkasnejši rok začetka zadnje aktivnosti Z

$$t_{LS}(Z) = t_{LF}(Z) - t(Z) \tag{5.13}$$

3. Najkasnejše roke dokončanja ostalih aktivnosti določimo po naslednji enačbi

$$t_{LF}(U) = \min_{V \in \mathcal{N}}(t_{LF}(V) - t(V)) \tag{5.14}$$

kjer je $\mathcal{N}$ množica vseh neposrednih naslednikov aktivnosti U.

4. Najkasnejši roki začetkov aktivnosti pa se izračunajo po enačbi

$$t_{LS}(X) = t_{LF} - t(X) \tag{5.15}$$

Primer analize kritične poti v aktivnostnem mrežnem diagramu za projekt, definiran s tabelo 5.1, je podan na sliki 5.8.

Časovne rezerve v aktivnostnem mrežnem diagramu

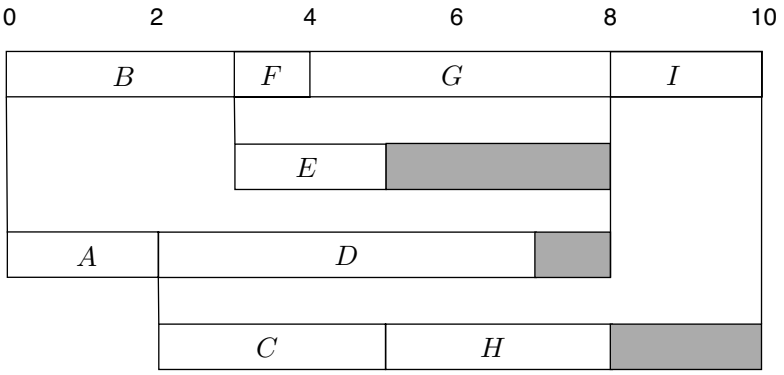
Drsenje aktivnosti X izračunamo po enačbi:

$$D(X) = t_{LS}(X) - t_{ES}(X) = t_{LF}(X) - t_{EF}(X). \tag{5.16}$$

Če je drsenje $D(X) = 0$, je aktivnost X kritična aktivnost.

5.2.3 Ganttovi diagrami

Informacijo o začetku in koncu aktivnosti je možno bolj pregledno podati s tako imenovanimi Ganttovimi diagrami. Aktivnosti v Ganttovih diagramih predstavimo kot pravokotnike, katerih dolžina predstavlja čas trajanja vzdolž vodoravne časovne osi. Za Ganttove diagrame je značilno, da so vse aktivnosti postavljene v najzgodnejši možni začetek. Na sliki 5.9 je Ganttov diagram za projekt s slike 5.8 oziroma slike 5.5. Ganttovi diagrami omogočajo hiter pregled vrstnega reda aktivnosti in njihovega trajanja. Ganttovi diagrami so koristni tudi za kontrolo med izvajanjem, saj lahko z barvanjem pravokotnikov označujemo stopnjo dovršenosti vsake aktivnosti, glede na navpično datumsko premico pa hitro ugotovimo, ali aktivnost kasni ali prehiteva. Na drugi strani pa je iz Ganttovih diagramov



Slika 5.9: Ganttov diagram za projekt s slike 5.8.

težko razbrati medsebojne odvisnosti aktivnosti in njihovih zmogljivosti. Sicer pa se da v Ganttove diagrame tudi vpisovati potrebne zmogljivosti za vsako aktivnost in medsebojno odvisnost aktivnosti (pred vsako aktivnost vpišemo številke tistih aktivnosti, ki se morajo dokončati pred njenim začetkom). Kljub temu se Ganttovi diagrami uporabljajo predvsem v kombinaciji z mrežnimi diagrami kot povzetek oziroma alternativen pogled na stanje projekta. Večina računalniških programov omogoča avtomatično kreiranje Ganttovih diagramov na osnovi mrežnega diagrama.

5.3 Analiza zmogljivosti

Dosedaj smo pri analizi projekta upoštevali le čas; zmogljivosti potrebne za izvajanje aktivnosti, kot so ljudje, stroji, orodja in surovine, pa smo obravnavali, kot da so na voljo v neomejenih količinah. Zato moramo po časovni analizi mrežni načrt uskladiti še z razpoložljivimi zmogljivostmi in specifičnimi časovnimi omejitvami celotnega projekta ali posameznih zmogljivosti. Temu usklajevanju pravimo obremenjevanje zmogljivosti (angl. loading of activities). Pri obremenjevanju zmogljivosti imamo lahko opravka z dvema vrstama omejitev. Ponavadi so zmogljivosti omejene in moramo aktivnosti prerazporediti ali izvajanje projekta celo nekoliko podaljšati, tako da nikoli ne presežemo v danem trenutku razpoložljivih zmogljivosti. Včasih pa je čas za dokončanje projekta omejen, tako da moramo z razporejanjem aktivnosti in čim manjšim dodajanjem zmogljivosti končati projekt v določenem časovnem okviru.

Če je veliko aktivnosti in veliko različnih vrst zmogljivosti, pride pri razvrščanju do ogromnega števila možnih kombinacij. Tudi računalniški

programi ne morejo pregledati in med seboj primerjati vseh možnih kombinacij in najti optimalne rešitve, saj bi bilo iskanje take rešitve prezamudno in predrago. Zato ne iščemo najboljših možnih rešitev, ampak le približno optimalne oziroma take, ki bodo *zadovoljile* naše potrebe. Način iskanja zadovoljive rešitve je iterativen proces s katerim postopoma izboljšujemo rešitev. Pri iskanju rešitve gre za prepletanje treh metod:

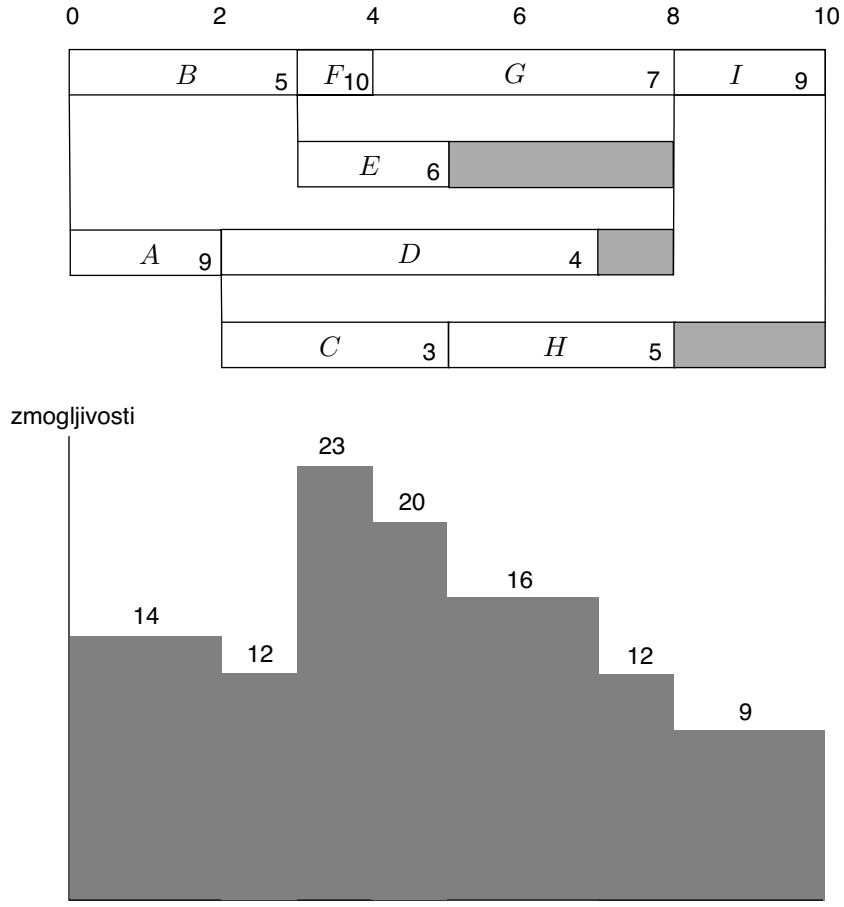
1. Pomemben element iskanja je to, da se že na samem začetku kar najbolj približamo iskani rešitvi, tako da z zdravorazumskim sklepanjem premeščamo aktivnosti v okviru njihovih časovnih rezerv. Tako razvrščanje aktivnosti lahko že zadosti zadanim omejitvam.

2. Drugi element razvrščanja zmogljivosti je tako imenovano sklepanje KAJ—ČE (what—if analysis), to je preizkušanje različnih scenarijev in primerjava njihovih posledic.

3. Le če ročno premeščanje ne da zadovoljivih rezultatov, uporabimo algoritme sestavljene iz različna hevristična pravil, ki teoretično sicer ne zagotavljajo optimalne rešitve, v praksi pa vodijo do sprejemljivih rešitev.

Razporeditev posameznih zmogljivosti grafično ponazorimo s histogrami (slika 5.10). K ustreznemu Ganttovemu diagramu dodamo histogram, narisan v istem časovnem merilu, kjer seštevamo potrebne zmogljivosti za vse aktivnosti, ki potekajo v istem časovnem intervalu. Običajno si prizadevamo razporediti zmogljivosti tako, da so le-te čimbolj enakomerno obremenjene ves čas trajanja projekta, kot je razvidno na sliki 5.11, saj so zmogljivosti običajno na voljo ves čas trajanja projekta. Če bi za isti primer imeli naenkrat na voljo le 14 enot zmogljivosti, bi morali projekt pač podaljšati.

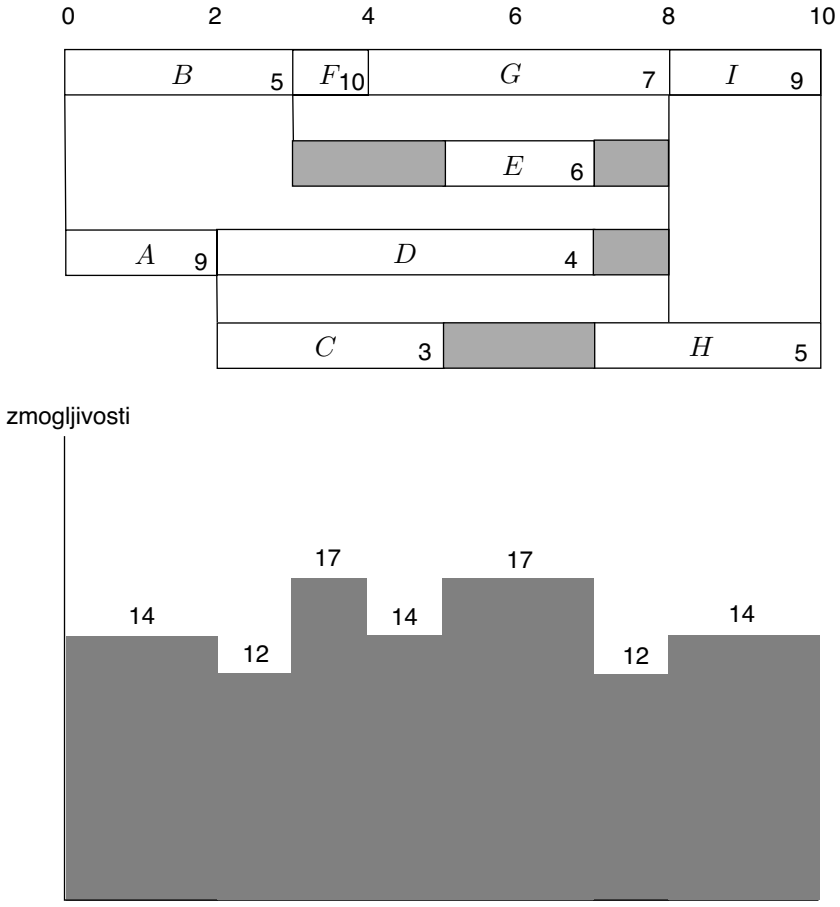
V nekaterih računalniških programih za mrežno planiranje pa lahko določimo točen urnik razpoložljivosti (po tednih, dnevih, celo urah) vsake posamezne zmogljivosti, ki ga program nato upošteva pri razporejanju. Pri avtomatskem razporejanju zmogljivosti računalniški programi uporabljajo algoritme, podobne enostavnim ročnim algoritmom [34]. Tako kot ročni algoritmi tudi tisti, vgrajeni v računalniške programe, ne morejo jamčiti globalne optimalne rešitve. Prednost računalniških programov pa je, da lahko med seboj usklajujejo veliko število aktivnosti in veliko različnih vrst zmogljivosti ter pri tem upoštevajo vse možne omejitve.



Slika 5.10: Ganttov diagram in histogram zmogljivosti za projekt, definiran v tabeli 5.1

5.4 Analiza stroškov

Planiranje stroškov vključuje vse aktivnosti, ki so predvidene za izvedbo oziroma uresničenje projektne naloge. Izhodišče za planiranje stroškov so ure, ki so predvidene za uresničenje posamezne aktivnosti, in planirani stroški za eno uro posamezne vrste dela. Če so v stroških (ceni) za eno uro dela vključeni vsi stroški, potem tako imenovanih direktnih stroškov ne planiramo posebej. Kadar pa imamo stroške ločene na tiste, ki so odvisni od števila opravljenih ur (indirektni stroški) in tiste, ki so neodvisni od njih (direktni stroški), tedaj moramo tudi slednje posebej planirati.



Slika 5.11: Ganttov diagram in histogram zmogljivosti za projekt, definiran v tabeli 5.1. Z razporejanjem aktivnosti znotraj časovnih rezerv je možno nekoliko izravnati zmogljivosti, ki so potrebne v posameznih časovnih enotah.

Med direktne stroške običajno sodijo stroški službenih potovanj (dnevnice, nočnine, prevozni stroški), stroški za udeležbo na seminarjih, posvetih ali razstavah, stroški za opravljeno delo ali storitve po pogodbi oziroma posebnem naročilu (strokovno svetovanje, pridobitev podatkov, izdelava programske opreme, nakup programske opreme), stroški za strokovno literaturo in druge materiale.

V indirektne stroške, poleg osebnih dohodkov s prispevki, sodijo tudi vsi ostali režijski stroški, ki so nastali v posamezni organizacijski enoti sku-

paj s stroški, ki bremenijo to organizacijsko enoto po raznih "ključih" za delitev stroškov uprave, prodaje, vzdrževanja in tako dalje. V primeru, ko ni na voljo podatkov o stroških za posamezno organizacijsko enoto, vzamemo povprečne letne stroške na enega delavca v višji organizacijski enoti, če pa teh stroškov ni niti na tej ravni, se zadovoljimo s povprečnimi stroški za enega delavca na nivoju podjetja. Te stroške nato delimo z letnim številom planiranih delovnih ur (med 1500 in 1800 urami). Ker pri projektni nalogi praviloma delajo delavci z nadpovprečno visokimi osebnimi dohodki in tudi režijskimi stroški (zlasti analitiki, programerji), povečamo tako izračunane povprečne stroške za eno uro dela z ustreznim faktorjem (na primer med 1.2 do 1.6). Kadar nimamo na voljo niti teh podatkov, si pomagamo s tržno ceno, ki jo za enaka dela uporabljajo tuje organizacije. V tem primeru moramo znižati to ceno za približno 20% do 40%, ker je v tržni ceni v takšni višini vračunano poleg dobička še določeno tveganje.

Planiranje stroškov po posameznih aktivnostih projekta nam omogoča, da lahko ugotavljamo stroške ne samo, ko je aktivnost končana, temveč tudi po posameznih dogodkih v mrežnem načrtu.

Na podoben način kot planiramo stroške, tudi zbiramo dejansko nastale stroške glede na dejansko opravljene ure po posamezni aktivnosti. Če razpolagamo z dejanskimi stroški za eno uro dela v posameznem časovnem obdobju (mesecu), pomnožimo dejansko opravljene ure s temi stroški, sicer pa s planskimi ali kalkulativnimi stroški za eno uro dela. Pri zbiranju tako imenovanih direktnih stroškov upoštevamo vedno samo dejansko nastale (obračunane ali zaračunane) stroške.

Primerjava med planiranim in dejanskim številom opravljenih ur pri posamezni aktivnosti ali vseh aktivnostih, ki imajo skupni končni dogodek, nam kaže natančnost planiranja in morebitne odklone — običajno so to prekoračitve. Primerjava med planiranimi in dejanskimi indirektnimi stroški nam kaže pravo sliko šele tedaj, če dejansko nastale indirektne stroške primerjamo s planiranimi za dejansko število opravljenih ur. Primerjava med planiranimi in dejansko nastalimi indirektnimi stroški nam da pravo sliko samo tedaj, če je planirano število ur enako dejanskemu številu opravljenih ur. V primerjavo lahko vključimo tudi direktne stroške, tako da jih prištejemo k indirektnim, čeprav ti stroški praviloma nastajajo precej neodvisno od dejanskega števila ur. Ko pa so končane vse aktivnosti, ki imajo isti skupni dogodek, je primerjava stroškov popolnoma zanesljiva.

Oglejmo si naslednji primer:

Aktivnost A			
Planirano	180 ur	Planirani indirektni stroški	600.000 SIT
Dejansko opravljeno	120 ur	Dejanski indirektni stroški	440.000 SIT

Iz primera vidimo, da je bilo opravljeno 66% planiranih ur, dejansko pa je bilo porabljeno že 73% planiranih stroškov. Očitno je, da stroški "prehitevajo" obseg opravljenega dela. Zato je potrebno ugotoviti razloge in v upravičenem primeru spremeniti plan indirektnih stroškov ali pa sprejeti kake druge ukrepe za odpravo tega nesorazmerja.

Planiranje in spremljanje stroškov se torej izvaja po aktivnostih, zbirno pa po dogodkih in za celoten projekt. Če se pokaže potreba, lahko izkazujemo stroške tudi po fazah projektnega dela. Opravljene ure, ki jih knjižimo k posameznemu projektu, in direktne stroške mora potrditi vodja projekta ali pooblaščena oseba.

Planiranje stroškov po posameznih projektih je pri nas slabo razvito, ker za to doslej največkrat ni bilo potrebe ali pritiska. Dosledna uvedba tržnih odnosov bo zahtevala pri načrtovanju projektov večjo varčnost in uvedbo ekonometrije.

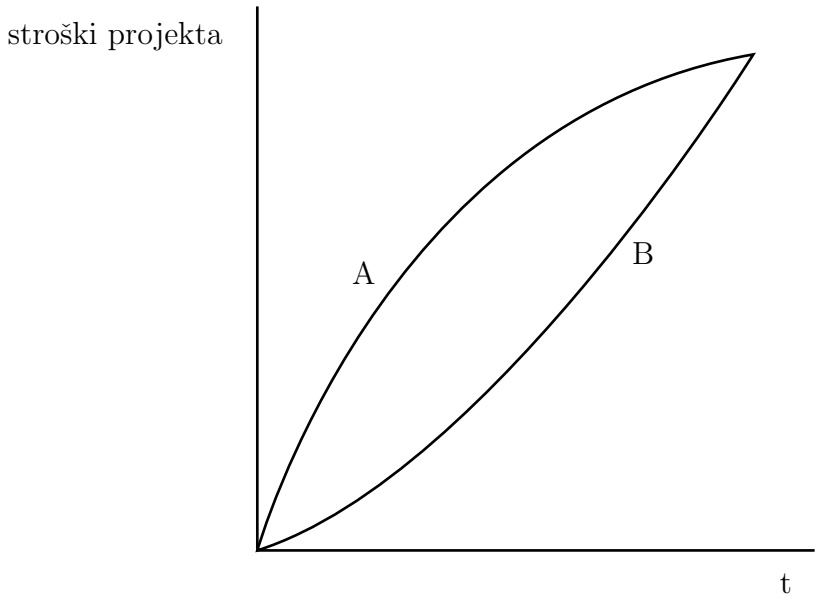
5.4.1 Metoda PERT/Cost

Metoda mrežnega planiranja PERT/Cost obravnava stroške na enak način kot zmogljivosti. Stroški, povezani z določeno aktivnostjo, niso vedno z njo tudi časovno sinhronizirani. Nekatere stvari plačujemo vnaprej ali pa za nazaj. PERT/Cost lahko prikaže take časovne zamike. Pri velikih projektih to lahko pomeni, da si moramo denar sposoditi in za posojilo plačevati obresti. Takrat se izplača aktivnosti, ki se jih da zamikati, začeti čim kasneje, saj smo tako v boljšem finančnem položaju (slika 5.12). Metoda PERT/Cost omogoča kontrolo velikih projektov glede na čas in stroške.

5.5 Skrajšanje trajanja projekta

Časovna analiza nam razkrije tiste aktivnosti, ki so kritične za izvajanje projekta. Čas trajanja projekta je torej določen z aktivnostmi na kritični poti. Če hočemo skrajšati izvajanje projekta, moramo skrajšati katero od kritičnih aktivnosti. Zato se splača bolj natančno preučiti kritične aktivnosti. V zvezi s kritičnimi aktivnostmi si zastavimo naslednja vprašanja:

- Kaj dosežemo z določeno aktivnostjo?



Slika 5.12: Če se aktivnosti začnejo prej (A), se stroški financiranja projekta večajo hitreje, kot če se aktivnosti začnejo kasneje (B).

- Kje aktivnost poteka?
- Kdaj je uresničena?
- Kdo jo opravlja?
- Kako je opravljena?

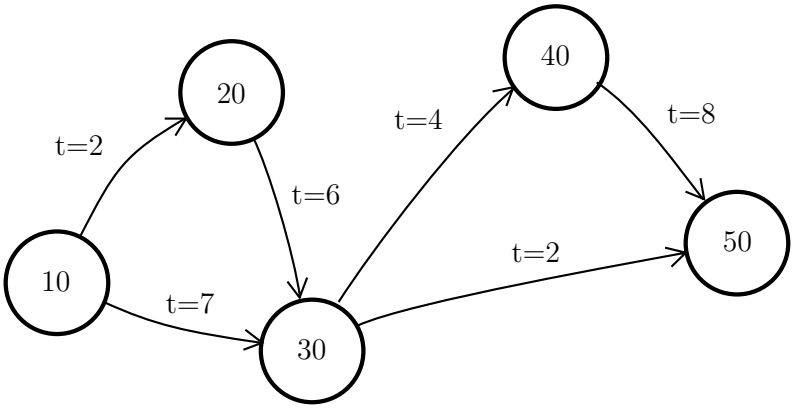
Odgovori na ta vprašanja nam pomagajo dodatno osvetliti aktivnost. Spremenimo lahko naše ocene glede njenega trajanja ali potrebe po zmogljivostih. Spremenimo pa lahko tudi medsebojne odvisnosti aktivnosti, saj lahko odkrijemo, da se neka aktivnost lahko začne, še preden se predhodna kritična aktivnost konča. Skratka, odkrijemo lahko alternativne načine za dosego istih ciljev. Običajno je skrajšanje aktivnosti povezano z večjimi stroški. Povezava med časom trajanja aktivnosti in njenimi stroški je predmet metode kritične poti.

5.5.1 Metoda kritične poti

Kadar hočemo skrajšati trajanje neke aktivnosti, moramo povečati zmogljivosti ali izbrati dražje rešitve. Imenujmo prvotno ocenjeni čas trajanja

aktivnosti *normalni čas* (t_n) in z njim povezane stroške *normalno ceno* (C_n). Za *izjemni čas* (t_i), to je minimalni realistični čas za dovršitev aktivnosti, pa so tudi stroški večji — *izjemna cena* (C_i). Čeprav funkcijska odvisnost med časom trajanja in stroški ni linearna, pa vseeno vse vmesne vrednosti aproksimiramo s premico, ki ima naslednji naklon:

$$\frac{C_i - C_n}{t_n - t_i} \tag{5.17}$$



Slika 5.13: Projekt, ki ga krajšamo z metodo kritične poti

Oglejmo si, kako lahko skrajšamo čas trajanja naslednjega projekta (slika 5.13) z metodo kritične poti.

Aktivnost	Normalni čas [tedni]	Izjemni čas [tedni]	Normalna cena [SIT]	Izjemna cena [SIT]	Naklon [SIT/teden]
10–20	2	1	150.000	170.000	20.000
10–30	7	5	120.000	200.000	40.000
20–30	6	4	240.000	300.000	30.000
30–40	4	3	40.000	60.000	20.000
30–50	2	1	400.000	420.000	20.000
40–50	8	6	150.000	250.000	50.000

Če hočemo skrajšati projekt za en teden, izbiramo med kritičnima aktivnostima 10-20 in 30-40, ki imata najmanjši naklon. Ker bi skrajšanje aktivnosti 10-20 povzročilo, da bi tudi aktivnost 10-30 postala kritična, najprej skrajšamo aktivnost 30-40. Kadar moramo skrajšati dve vzporedni aktivnosti, moramo njuna naklona sešteti.

Večanje cene celotnega projekta pri nadaljnjem krajšanju projekta je podano v spodnji tabeli.

	Cena [SIT]	tedni
Normalni čas	1.100.000	20
Normalni čas – 1 teden	1.120.000	19
Normalni čas – 2 tedna	1.140.000	18
Normalni čas – 3 tedni	1.190.000	17
Normalni čas – 4 tedni	1.240.000	16
Normalni čas – 5 tednov	1.310.000	15
Normalni čas – 6 tednov	1.380.000	14

Pri metodi kritične poti nismo razločevali med vrstami stroškov. To je bila precej groba poenostavitev, stroške bi morali obravnavati diferencirano, tako kot to opisuje predhodni razdelek.

5.6 Kontrola izvajanja

Mrežni načrt se izdela v fazi definiranja projekta, uporabljamo pa ga predvsem v fazi izvajanja projekta. S pomočjo mrežnega načrta primerjamo izvedeno delo in dejanske stroške z načrtovanim delom in načrtovanimi stroški. Napovedujemo lahko nadaljnje delo in se, če je to potrebno, odločamo za korektivne akcije.

Kontrola izvajanja projektov se običajno izvaja ob vnaprej določenih mejnikih (angl. milestones). Takrat zberemo vse relevantne informacije (presek stanja), skličemo sestanek, kjer se izvajanje projekta analizira in predlaga morebitne ukrepe.

5.7 Vloga računalnikov v projektnem delu

Narava načrtovanja projektov tako rekoč kliče po uporabi računalnikov. Računalniki so se sicer res začeli uporabljati za mrežno načrtovanje že v 50-tih letih. Vendar so bili tedaj še izredno dragi in dostopni le državnim ustanovam in največjim podjetjem. Zato so se razvili ustrezni ročni postopki mrežnega načrtovanja [34]. Ročno mrežno planiranje je seveda že samo po sebi zamudno, njegova največja slabost pa je, da je nekfleksibilno. Take mrežne načrte je težko in zelo zamudno spreminjati in jih sproti usklajevati z dejanskim potekom izvajanja projekta. Zaradi teh slabosti se za manjše projekte niti ni izplačalo izdelati mrežnih načrtov, saj so bili stroški načrtovanja preveliki. Pri večjih projektih, kjer se je sicer izdelal mrežni načrt z ročnimi

metodami, pa načrt velikokrat ni igral zadostne vloge med samim izvajanjem projekta, saj je bilo vnašanje sprememb, do katerih med izvajanjem večine projektov nedvomno pride, prezamudno.

Ko so se računalniki nekoliko bolj razširili in so se pojavili tudi v naših podjetjih, se je položaj le nekoliko izboljšal. Programi za projektno načrtovanje na teh računalnikih so bili namreč za uporabo dokaj zapleteni. Tako kot večina takratne programske opreme so zahtevali paketno obdelavo in podporo sistemskih operaterjev oziroma programerjev. Računalniška obdelava projektnega načrta se je morala izdelati v računalniškem centru podjetja, ne pa tam, kjer se je dejansko načrtovalo, še manj pa tam, kjer se je projekt izvajal. Zaradi prostorske in časovne neenotnosti ter potrebe po posrednikih med dejanskimi načrtovalci in programi je še vedno šepala podpora pri vodenju projektov, saj je bila pot do vnašanja sprememb še vedno zamudna in zapletena.

Prihranki in prednosti načrtovanja in vodenja projektov s pomočjo računalniških programov:

- Direktni prihranki so zaradi natančnejšega ocenjevanja zmogljivosti in stroškov.
- Posredni prihranki so zaradi:
  - zmanjšanja zalog,
  - zmanjšanja kapitalnih stroškov zaradi boljše izkoriščenosti naložb,
  - zmanjšanih stroškov poslovanja.
- Težko merljivi posredni prihranki pa so zaradi:
  - boljše kvalitete izdelkov,
  - boljše oblikovanih zalog in s tem večje dosegljivosti izdelkov,
  - skrajšanja dobavnih rokov,
  - zadovoljnejših kupcev.

S pojavom cenenih in vsem dostopnih osebnih računalnikov pa se je pri načrtovanju in vodenju projektov, tako kot na številnih drugih področjih uporabe računalnikov, veliko spremenilo. Poleg tega, da je postalo načrtovanje z računalnikom veliko cenejše, je postala tudi uporaba programske opreme veliko enostavnejša. Projektno načrtovanje z računalnikom je zato postalo ekonomsko upravičeno tudi za manjše projekte in za manjša podjetja. S programom za mrežno načrtovanje pa je lahko operirala sama pro-

jektna skupina, tako v fazi načrtovanja kot tudi v fazi izvajanja projekta. Zato je šele tedaj postalo praktično dosledno sprotno spremljanje izvajanja projekta in odločanje o spremembah s pomočjo računalniških programov.

5.8 Kriteriji za ocenjevanje programske opreme

Programe za načrtovanje in vodenje projektov ocenjujemo predvsem po naslednjih kriterijih:

- katere so možne povezave oziroma odvisnosti med aktivnostmi (konec – začetek, začetek – konec, začetek – začetek, konec – konec),
- vrsta mrežnega načrta (aktivnostni ali dogodkovni),
- način spremljanja projektov med izvajanjem (kako je možno primerjati dejansko stanje z načrtovanim),
- kakšno je obremenjevanje zmogljivosti (kako podrobno, ali je možno avtomatično obremenjevanje zmogljivosti, določanje in kontrola stroškov in urnikov za posamezne zmogljivosti),
- kakšen je uporabniški vmesnik med programom in uporabnikom (način vnašanja in spreminjanja podatkov),
- kakšna je podpora poročanju in kakšna je vizualizacija vseh informacij?
- ali je možno podatke izmenjevati z drugimi aplikacijami?

Številni uporabniki programov za računalniško načrtovanje in vodenje na osebnih računalnikih niso redni uporabniki računalnikov ali se z računalnikom celo prvič srečajo. Zato je vloga primernega uporabniškega vmesnika izrednega pomena za lažjo uporabo in hitrejše učenje. Primerni so predvsem taki vmesniki, kjer je možno vnašanje in spreminjanje podatkov s pomočjo miške preko grafičnih prikazov.

5.8.1 Program SuperProject

Na platformi IBM-PC sta v Sloveniji najbolj pogosta programa za mrežno načrtovanje *SuperProject* in *Microsoft Project*. Po svojih funkcijah in zmogljivosti sta si precej podobna.

Glavne lastnosti programa SuperProject [31]:

- možne povezave med aktivnostmi so: konec – začetek, začetek – konec, začetek – začetek, konec – konec,
- avtomatično obremenjevanje zmogljivosti (tudi med več projekti),
- pripisovanje fiksnih in variabilnih stroškov posameznim aktivnostim, zmogljivostim in projektom,
- različni pogledi na projekt: aktivnostni diagram (PERT), Ganttovi diagrami, pregled strukture projekta, spiski aktivnosti, zmogljivosti, vključno z urniki za posamezne zmogljivosti.

Ukaze je možno izbirati preko menujev, ki se nahajajo na vrhu ekrana, bodisi s pomočjo miške ali tipkovnice. Kateri ukazi so v danem trenutku možni, je odvisno od tega v katerem od štirih možnih pogledov na projekt se nahajamo in od stopnje zahtevnosti. SuperProject namreč omogoča dvostopenjsko nastavljanje zahtevnosti. Lažja stopnja je namenjena začetnikom ali neizkušenim uporabnikom in nezahtevnim projektom.

SuperProject nudi štiri glavne vrste pogledov na projekt:

- glavna preglednica projekta,
- aktivnostni (PERT) diagram,
- ganttovi diagrami,
- struktura projekta (diagram WBS).

Načrtovanje kot ponavadi začnemo z definiranjem ciljev, posameznih aktivnosti in ugotavljanjem povezav med aktivnostmi. Namesto na papirju lahko te podatke vnašamo neposredno bodisi v glavno preglednico projekta ali v aktivnostni diagram. V **glavno preglednico** ali tabelo vnesemo oziroma iz nje razberemo ime in oznako posamezne aktivnosti, zmogljivosti, potrebne za to aktivnost, prioriteto aktivnosti, trajanje aktivnosti, načrtovan začetek ter konec aktivnosti, pa tudi odvisnosti med aktivnostmi. V nekatera polja v tabeli lahko informacije vnašamo direktno, v drugih poljih pa so prikazane informacije, ki so odvisne od drugih in so iz njih avtomatično izračunane (na primer začetki in konci aktivnosti).

Tudi v **aktivnostni diagram** lahko vnesemo iste podatke, le da se na tem diagramu neposredno vidi odvisnost aktivnosti med seboj. S pomočjo miške lahko kreiramo nove aktivnosti, jih premikamo po zaslonu, jim določamo potrebne zmogljivosti, čas trajanja in jih povezujemo z drugimi aktivnostmi. Avtomatično se izračuna kritična pot, posebej so označene tudi kritične aktivnosti.

Tretji možni pogled na projekt je pogled preko **Ganttovih diagramov**, ki omogoča direkten pregled nad trajanjem projekta. Tudi v Ganttovem diagramu je možno vnašati oziroma spreminjati podatke o posameznih aktivnostih. Posebna vrsta Ganttovih diagramov so tudi histogrami za posamezne zmogljivosti. Iz teh diagramov lahko razberemo, kako so v toku trajanja projekta obremenjene posamezne zmogljivosti ali ljudje.

SuperProject omogoča definiranje projektov na devetih hierarhičnih nivojih. Aktivnost na določenem nivoju sestavlja več aktivnosti na nižjem nivoju. Informacije z nižjega nivoja se avtomatično prenašajo na višji nivo, tako da se seštevajo časi, zmogljivosti in stroški. Pregled nad hierarhično strukturo projekta omogoča **diagram WBS**, ki prikaže drevesno strukturo vseh aktivnosti.

Poleg štirih glavnih pogledov na projekt se v posameznih tabelah avtomatično zbirajo podatki za celoten projekt, po posameznih aktivnostih in po posameznih zmogljivostih. Za vsako zmogljivost lahko določimo njeno ceno in urnik razpoložljivosti, ki ga program upošteva pri razvrščanju in računanju stroškov. Med izvajanjem projekta je možno poleg načrtovanih informacij (časovni roki, stroški) vnašati še dejanske. Program tudi avtomatično kreira poročila o projektu, ki jih sestavi iz posameznih tabel in vseh drugih informacij, ki jih ima o projektu.

5.9 Mrežno načrtovanje v informacijskem sistemu podjetja

Računalniško podprto načrtovanje in vodenje projektov je sedaj praktično dostopno vsakomur. Za manjše projekte je pri računalniškem vodenju pomembna predvsem organizacijska plat. Podrobna analiza stroškov, ki je možna z večino programov, za manjše projekte ni tako bistvena, kot je za velike projekte.

S pojavom računalniškega vodenja posameznih projektov se zastavi vprašanje povezovanja več projektov in posredovanja informacij o posameznih projektih v celotni informacijski sistem podjetja.

Nekateri programi za načrtovanje in vodenje projektov, med njimi je tudi *SuperProject*, sicer omogočajo delitev skupnih zmogljivosti med več projekti. Toda problem povezovanja se kaže širše, še posebej če je potrebno posredovati informacije o projektih na nivo poslovnega in strateškega odločanja nekega podjetja. Zato je pomembna enostavnost posredovanja informacij iz programov za projektno delo raznim drugim programom za finančno načrtovanje, strateško planiranje in odločanje. Obstajajo celo ekspertni sis-

temi, ki načrtovalcem svetujejo pri načrtovanju in izvajanju projektov glede na obstoječe razmere.

S fizičnega oziroma strojnega vidika se ta problematika kaže z uvajanjem računalniških mrež in priključevanjem osebnih računalnikov na velike računalniške sisteme in izmenjevanjem informacij.

Poglavje 6

Posebnosti projektov za razvoj programske opreme

Na začetku so pri načrtovanju in razvoju programske opreme enostavno prevzeli izkušnje in prakso z drugih področij, predvsem izkušnje gradnje strojne opreme. Vendar imajo projekti razvoja programske opreme veliko posebnosti. Še celo med seboj se zelo razlikujejo glede na to, kako so organizirani, vodeni in kakšne cilje imajo. Pri nekaterih projektih so stroški vnaprej določeni in kvaliteto končnega produkta je potrebno maksimizirati glede na to omejitev. Za druge pa je vnaprej predpisana kvaliteta programske opreme in projekt mora učinkovito zgraditi sistem, ki bo zadostil tem kriterijem kvalitete. Tretji spet morajo biti končani v minimalnem možnem času. Vodja projekta lahko vpliva na izvajanje projekta z izbiro metod in orodij za razvoj, organizacijo projekta in ravnijo učinkovitosti rezultirajoče programske opreme. Zunanjih parametrov (npr. izkušenj uporabnikov, števila ljudi v projektni skupini) pa ni možno spreminjati.

Delitev na ciljne, kontrolne in zunanje parametre ni stalna, saj je lahko nek parameter enkrat zunanji, drugič kontrolni ali ciljni. Če so cilji projekta (čas razvoja, kvaliteta) točno določeni, moramo zaposliti ustrezne ljudi in nabaviti potrebna orodja. Če je delovna skupina fiksna, pa lahko kontroliramo projekt z daljšanjem časa razvoja ali manjšanjem kvalitete programske opreme. Za uspešno vodenje projekta je bistveno, da je vnaprej znano, katere parametre je možno spreminjati in z njimi kontrolirati potek projekta.

Vendar pa je v praksi težko natančno kontrolirati izvajanje projektov, saj je učinke kontrolnih akcij žal težko natanko predvideti. Tako moramo namesto sistemsko zasnovane kontrole uporabljati bolj intuitivne in primitivne metode, ki slonijo predvsem na izkušnjah.

6.1 Vrste projektov za razvoj programske opreme

Projekte za razvoj programske opreme lahko razdelimo na več kategorij glede na naslednje tri značilnosti [23]:

Stabilnost produkta je odvisna od tega, kako natančno so definirane zahteve po funkcionalnosti in kvaliteti.

Stabilnost razvojnega procesa je odvisna od stopnje kontrole, ki jo imamo nad razvojem, in od natančnosti, s katero lahko merimo razvoj.

Stabilnost zmogljivosti določa predvsem razpoložljivost kvalificiranih razvijalcev.

Tako je na primer stabilnost produkta visoka, če so zahteve jasne. Če pa je potrebno zahteve šele ugotoviti in/ali če se zahteve pogosto spreminjajo, je stabilnost produkta nizka. Če ima vsaka od treh skupin značilnosti visoko ali nizko stabilnost, dobimo osem možnih kombinacij. Nekatere od teh osmih kombinacij niso realistične, saj če je stabilnost produkta nizka, težko pričakujemo, da bo stabilnost razvojnega procesa in zmogljivosti visoka.

Tabela 6.1: Štiri osnovne vrste projektov razvoja programske opreme glede na problematiko vodenja

	Stabilnost		
Vrsta problema	produkta	razvoj. procesa	zmogljivosti
Problem realizacije	visoka	visoka	visoka
Problem razvrstitve	visoka	visoka	nizka
Problem načrtovanja	visoka	nizka	nizka
Problem iskanja	nizka	nizka	nizka

Zanimive so predvsem štiri kombinacije, prikazane v tabeli 6.1:

1. **Problem realizacije.** Če so zahteve jasne in stabilne, če vemo, kako naj teče razvojni proces in če imamo za povrh še dovolj zmogljivosti, lahko razvojni proces natančno kontroliramo. V tej skoraj idealni situaciji se lahko osredotočimo le na problem realizacije programske opreme na najbolj učinkovit način. Uporabimo lahko kar klasični razvojni cikel, za koordinacijo dela pa direktni nadzor. Delo se lahko vodi na ločitveni način, saj so naloge natančno določene s pomočjo pravil in procedur.

2. **Problem razvrstitve.** Ta problem se od prvega razlikuje po tem, da nimamo dovolj zmogljivosti. Tu so mišljeni predvsem kvalificirani razvijalci. Osnovni problem je zato ta, kako razvijalce razvrstiti na delovne aktivnosti. S čim večjo standardizacijo razvojnega procesa je možno povečati izmenljivost ljudi med nalogami, kar je v taki situaciji izredno pomembno. Tudi v tem primeru lahko izberemo klasični razvojni cikel.

3. **Problem načrtovanja.** Če so zahteve jasne in stabilne, ne vemo pa, kako jih naj uresničimo in kakšne zmogljivosti potrebujemo, gre za problem načrtovanja projekta. Ugotoviti moramo, kateri so glavni mejniki v projektu, katero dokumentacijo moramo pripraviti in kdaj, koliko ljudi potrebujemo in kako bomo vse skupaj kontrolirali. Primerna je delitvena oblika organizacije dela, kjer koordinacija poteka s pomočjo standardizacije delovnih rezultatov. Ker so rezultati definirani, se lahko kontrola potemtakem omeji le na izvajanje aktivnosti in določanje potrebnih zmogljivosti. Za lažje vodenje potrebujemo nekaj več zmogljivosti in možnost prekoračitve časa in stroškov. Kot razvojni model je nabolj primeren postopni razvoj.

4. **Problem iskanja.** Če so zahteve nejasne, ni mogoče definirati niti razvojnega postopka niti določiti razvojnih zmogljivosti. Tak projekt bo potem razvojne narave, kjer bomo pravo rešitev šele sproti iskali. V tem primeru je najboljša organizacijska oblika pripadnost skupini, kjer je koordinacijski mehanizem vzajemno prilagajanje. Kontrola takega projekta je zelo težavna, kot razvojni model pa je primeren razvoj s pomočjo serije prototipov.

6.2 Zrelostne stopnje razvojne skupine

Humphrey [25] je predlagal lestvico petih zrelostnih stopenj skupin za razvoj programske opreme:

1. **Začetna** stopnja zrelosti pomeni, da razvojna skupina deluje brez formalnih postopkov, brez projektnih načrtov, brez ocenjevanja stroškov — skratka *ad hoc*. Skupina lahko napreduje na naslednjo stopnjo,

   - če vzpostavi projektni način vodenja razvoja,
   - če poskrbi za kontrolo kvalitete, tako da se vse dela, kot je treba in

- da vzpostavi sistem za upravljanje s konfiguracijami.

2. **Ponovljiva** stopnja zrelosti pomeni, da skupina na kontroliran način določa načrte in aktivnosti. Da bi lahko napredovala na naslednji zrelostni nivo, pa mora:

   - ustanoviti posebno skupino za iskanje izboljšav razvojnega procesa,
   - vzpostaviti okvirno razvojno arhitekturo, znotraj katere se definirajo specifični projekti,
   - uvesti družino metod, orodij in standardov za načrtovanje, kodiranje in testiranje programske opreme.

3. **Določljiva** stopnja zrelosti pomeni, da ima skupina dovolj dobre temelje, da se lahko stalno izpopolnjuje. Glavni koraki do naslednje stopnje zrelosti so:

   - merjenje in shranjevanje podatkov o poteku razvoja projektov,
   - ambiciozen načrt za zagotavljanje kvalitete programske opreme s pomočjo kvantitativnih meritev.

4. **Vodljiva** stopnja zrelosti pomeni, da se kvantitativni podatki o poteku razvoja rutinsko zbirajo in analizirajo. Za naslednjo stopnjo je zato pomembno:

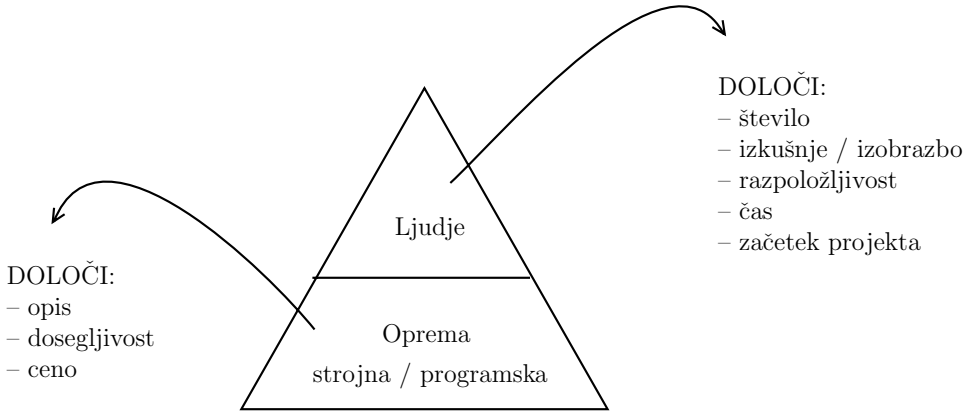
   - avtomatično zbiranje podatkov,
   - uporaba teh podatkov za analizo in spremembo razvojnega procesa, da bi preprečila nastanek problemov.

5. **Optimizirajoča** stopnja zrelosti pomeni stabilno osnovo za izboljšave samega procesa razvoja. Medtem ko je pozornost na nižjih nivojih osredotočena na izboljšave produkta, je tu v središču pozornosti sam proces razvoja.

Ko je Humphrey leta 1989 sestavil to lestvico, je še raziskal, na kateri stopnji se nahajajo razvojne skupine v ZDA. Večina jih je bila na začetni, nekaj na ponovljivi, zelo redke pa na določljivi stopnji. Nobena skupina ali projekt ni deloval na vodljivi in optimizirajoči zrelostni stopnji.

6.3 Ocenjevanje stroškov

Načrtovalec projektnega načrta ima pred seboj podobno nalogo kot pisatelj, ki želi oceniti, koliko časa bo pisal svoj naslednji roman, za katerega je izbral le določeno temo. Tako kot pri romanu je tudi pri programski kodi osnovni parameter, ki vpliva na čas pisanja, dolžina romana oziroma število programskih vrstic. Bolj ko bo pisatelj razdelal svoj načrt, roman razdelil na poglavja in določil glavne osebe, lažje bo ocenil obseg celotnega dela. Tudi ocenjevanje obsega programske kode postaja lažje, ko napreduje analiza in načrtovanje kode. Poleg obsega kode pa na čas pisanja vplivajo še številni drugi faktorji, kot so izkušnje pri podobnem delu, zahtevnost kode in zunanje okoliščine. Težava ocenjevanja pa je, da si *čimbolj točne informacije želimo čim hitreje*.



Slika 6.1: Zmogljivosti, ki jih moramo določiti med izdelavo projektnega načrta.

Za izdelavo projektnega načrta za razvoj programske opreme moramo najprej vedeti:

1. kaj so zahteve oziroma cilji (funkcije programske opreme, število uporabnikov, hitrost odziva, zanesljivost),

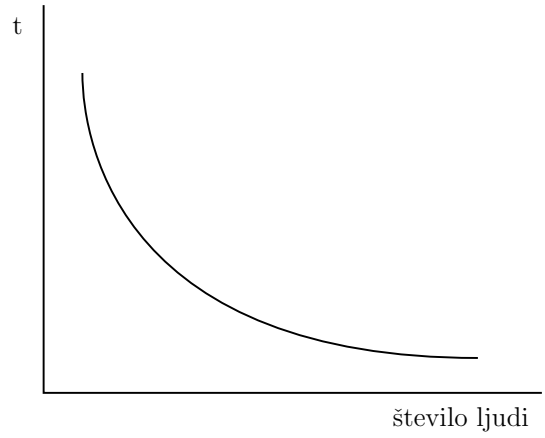
2. nato pa moramo oceniti potrebne zmogljivosti, čas in stroške za dosego ciljev (slika 6.1).

Razvoj in načrtovanje programske opreme je pretežno intelektualno delo, ki ga je že tako ali drugače težko meriti, relativna mladost programskega inženirstva in s tem povezano pomanjkanje izkušenj pa še dodatno otežuje

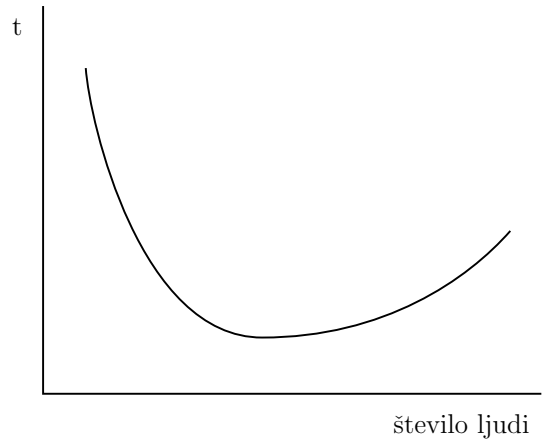
ocenjevanje dela. Projektno načrtovanje se mora zato zadovoljiti pogostoma le z grobimi in okvirnimi ocenami, ki jih lahko kasneje med potekom projekta dopolnjujemo. Večja ko je kompleksnost in velikost naloge, večja je nezanesljivost ocen. Težje je tudi ocenjevati in kontrolirati daljše projekte, saj izkušnje kažejo, da je se sodelavci pri večletnih projektih predolgo "ogrevajo", na koncu pa hitijo, saj zmanjkuje časa. Pri več let trajajočih projektih je težava tudi izredno hitro se spreminjajoča tehnologija in navsezadnje tudi koncepti razvoja programske opreme. Vodstvo projektov razvoja programske opreme se mora zato zavedati, kakšna stopnja natančnosti in zanesljivosti je v danem primeru sploh mogoča, da ne išče natančnosti tam, kjer je možna le groba ocena.

6.3.1 Merila za ocenjevanje

Da lahko nekaj ocenjujemo ali merimo, se moramo najprej dogovoriti za ustrezno merilo. Razvoj programske opreme je delovno intenzivno, daleč najpomembnejša zmogljivost so ljudje z ustreznim znanjem in izkušnjami. Običajno merilo za opravljeno delo—**[človek − mesec]** je v teh okoliščinah zelo varljivo merilo [9]. Programiranje je kompleksno delo, kjer učenje in interakcije med ljudmi igrajo veliko vlogo. V svetu na splošno primanjkuje izkušenih in produktivnih programerjev. Če upoštevamo še ocene, da se produktivnost programerjev lahko razlikuje med seboj celo v razmerju $1 : 10$, lažje razumemo, da je ocenjevanje števila potrebnih programerjev zelo kočljiva naloga.



Slika 6.2: Za razcepitveno delo je značilno, da več ljudi hitreje konča določeno delo.



Slika 6.3: Pri delu, kjer moramo usklajevati delo posameznikov, je značilno, da se razmerje med številom ljudi in potrebnim časom pri določenem številu ljudi obrne. Večanje števila ljudi ne skrajšuje več potrebnega časa, ampak ga celo podaljšuje.

Število programerjev in čas za delo nista direktno zamenljiva (primerjaj sliki 6.2 in 6.3). Za razcepitvneno delo, ki zahteva dodatno komuniciranje, obstaja neko optimalno število ljudi, ki se še lahko ustrezno dogovarjajo med seboj in opravijo delo v najkrajšem možnem času. Če bi povečali to število, bi več časa porabili za komuniciranje in učenje, tako da bi se potrebni čas za dokončanje dela lahko celo podaljšal (slika 6.3). Dejstvo, da storilnost posameznih programerjev v večjih skupinah pada, je dobro znano.

Manjše število načrtovalcev programske opreme je zaželeno tudi z vidika konceptualne enotnosti. Več ljudi sicer več ve, toda programska oprema ni le skupek dobrih idej in rešitev, temveč organska celota, katera vrednost se ne meri z genialnostjo posameznih podrobnosti, ampak s tem, kako se posamezni deli ujemajo in tvorijo funkcionalno celoto. Tako konceptualno enotnost je najlažje doseči, če je programska oprema produkt mišljenja ene same osebe. Razvoj programske opreme bi se sicer neupravičeno zavlekel, če bi prav vse delo moral opraviti posameznik. Zato se je na področju razvoja programske opreme uveljavila organizacijska oblika skupine glavnega programerja [9]. Tako kot kirurg ali pilot tudi glavni programer osebno in v celoti odgovarja za rezultate dela celotne ekipe. Vsi drugi člani ekipe mu pri tem pomagajo in svetujejo ter opravljajo vsa rutinska dela.

"Človek – mesec" kot merilo dela so zato lahko varljivi. Druga možnost za ocenjevanje velikosti projekta je **število programskih vrstic** (LOC

– Lines Of Code). Pri ocenjevanju gre torej za to, da vnaprej ocenimo, koliko programskih vrstic bo obsegala koda končanega projekta. Tudi to merilo ima pomanjkljivosti, saj je odvisno od vrste programskega jezika, vsak programer pa tudi lahko do določene mere poljubno raztegne ali skrči kodo. Nekoliko bolj zanesljivo merilo je namesto števila vrstic število operacij in operandov, ki nastopajo v programu. S pomočjo storilnosti (število programskih vrstic, ki jih programer v povprečju napiše v mesecu dni) je možno enostavno pretvarjati "človek – mesece" v število programskih vrstic in obratno.

Tretja vrsta meril, ki se uporablja za ocenjevanje v projektih, pa so **funkcijske točke**. Namesto štetja programskih vrstic skušamo obseg kode oceniti s točkovanjem različnih notranjih struktur (število in vrsta različnih podatkovnih struktur, različnih funkcij) ali zunanjih parametrov načrtovane programske opreme (število in vrsta vhodnih in izhodnih podatkovnih tipov, vmesnikov in kontrolnih struktur). Funkcijske točke so subjektivno merilo, ki ga tudi, ko je programska oprema napisana, ni možno nedvoumno preveriti, saj nimajo fizičnega ekvivalenta.

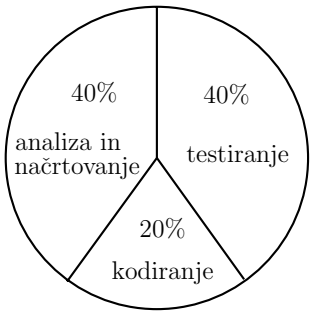
6.3.2 Metode za ocenjevanje

Razvojne skupine pogosto delajo napako, da sploh ne poskušajo sistematično oceniti potrebnega dela. Izhajajo iz nekih zunanjih omejitev, na primer koliko časa imajo maksimalno na voljo ali koliko lahko največ dobijo plačano za neko delo. Nekateri so celo prepričani, da se jim z delom tako mudi, da se enostavno nimajo časa ukvarjati z ocenjevanjem in načrtovanjem. Taka izhodišča za ocenjevanje dela skupaj s pomanjkanjem izkušenj imajo lahko katastrofalne posledice.

S pomočjo izbranih meril moramo oceniti obseg potrebnega dela pri razvojnem projektu. Za grobo in izhodiščno oceno je koristen podatek, kakšna je tipična delitev stroškov oziroma časa med glavnimi aktivnostmi v razvojnem ciklu (slika 6.4).

Pri ocenjevanju potrebnih zmogljivosti in časa za razvoj programske opreme se uporabljajo dekompozicijske in empirične metode [51].

Dekompozicijske metode podobno kot pri načrtovanju projektnega dela običajno razdelijo celotno potrebno delo na smiselne enote. Izvajanje teh delovnih enot lahko organiziramo v okviru posameznih projektnih aktivnosti. V tem primeru so delovne enote posamezni programski moduli (podprogrami, funkcije, vmesniki ipd.). Te manjše enote so bolj pregledne in jih zato lažje ocenimo bodisi s številom potrebnih



Slika 6.4: Tipična delitev dela pri razvoju programske opreme

programskih vrstic ali s funkcijskimi točkami. S pomočjo prevzetih ali lastnih, na osnovi že končanih projektov izpeljanih meril produktivnosti (npr. [*število programskih vrstic, človek – mesec*]), nato ocenimo potrebni čas in zmogljivosti.

Empirične metode za določanje zmogljivosti in časa uporabljajo izkustvene formule, izpeljane na osnovi dokončanih projektov. Formule povezujejo zmogljivosti s potrebnim delom in časom. Uteži v formulah pa določimo iz tabel, kjer točke odražajo zahtevnost in funkcionalnost programov.

Dekompozicijske metode lahko uporabimo takrat, ko že poznamo notranjo strukturo programske opreme. Empirične metode pa lahko uporabimo že bolj zgodaj v razvoju le na osnovi zahtev in omejitev.

V obeh primerih ocenjevanja potrebujemo strokovnjake, ki bodo podali svoje ocene. Da bi preprečili neutemeljen vpliv posameznega strokovnjaka na druge, je koristna metoda Delphi. Ta metoda zahteva koordinatorja, ki od vseh ocenjevalcev zbere njihove ocene. Te ocene (brez imen njihovih avtorjev) nato razdeli med ocenjevalce. Na osnovi ocen drugih lahko vsak modificira svojo oceno. V več krogih ocene ponavadi konvergirajo v dober približek.

Druga možna metoda za zanesljivejše določanje ocen pa zahteva od vsakega ocenjevalca, da poda optimistično, realistično in pesimistično oceno. Pričakovani rezultat izračunamo na enak način kot pri metodi PERT na strani 5.1.

6.3.3 Linearni modeli

Prvi modeli za ocenjevanje stroškov projekta razvoja programske opreme so bili linearni [44]. Linearni modeli imajo naslednjo obliko:

$$E = a_0 + \sum_{i=1}^{n} a_i x_i \tag{6.1}$$

kjer so a_i, $i = 0, \ldots, n$ konstante, x_i, $i = 1, \ldots, n$ pa faktorji, ki vplivajo na stroške. Faktorjev, ki vplivajo na stroške projekta, je cela vrsta. Na osnovi analize preteklih projektov je Nelson [44] predlagal model, ki vsebuje 14 faktorjev:

$$\begin{aligned} E \;=\; & -33.63 + 9.15x_1 + 10.73x_2 + 0.51x_3 + 0.46x_4 + 0.40x_5 + \\ & 7.28x_6 - 21.45x_7 + 13.5x_8 + 12.35x_9 + 58.82x_{10} + \\ & 30.61x_{11} + 29.55x_{12} + 0.54x_{13} - 25.20x_{14} \end{aligned} \tag{6.2}$$

V tej enačbi E pomeni ocenjeno število *človek – mesecev*. Obseg možnih vrednosti faktorjev x_i pa je podan v tabeli 6.2.

Tabela 6.2: Faktorji v Nelsonovem modelu [44]

Faktor	Opis	Možne vrednosti
x_1	Nestabilnost zahtev	0–2
x_2	Nestabilnost načrta	0–3
x_3	Odstotek matematičnih ukazov	procent
x_4	Odstotek vhodno-izhodnih ukazov	procent
x_5	Število podprogramov	številka
x_6	Uporaba višjih programskih jezikov	0 (da) / 1 (ne)
x_7	Poslovna aplikacija	0 (da) / 1 (ne)
x_8	Samostojni program	0 (da) / 1 (ne)
x_9	Prvi program na določeni vrsti računalnikov	1 (da) / 0 (ne)
x_{10}	Hkratni razvoj strojne opreme	1 (da) / 0 (ne)
x_{11}	Uporaba naprav z naključnim pristopom	1 (da) / 0 (ne)
x_{12}	Različna razvojni in ciljni računalnik	1 (da) / 0 (ne)
x_{13}	Število poslovnih poti	število
x_{14}	Razvoj za obrambno organizacijo	0 (da) / 1 (ne)

Čeprav je smer vpliva (povečanje ali zmanjšanje) posameznih faktorjev na obseg dela pravilen, pa je očitno, da ima na vrednosti konstant in faktorjev odločujoč vpliv zbirka projektov, na katerih je bil model razvit. S

povsem merskega vidika tudi ni korektno, da v istem modelu seštevamo cele vrednosti, procente in Boolove vrednosti. Malo je tudi verjetno, da imajo vsi našteti faktorji linearen in neodvisen vpliv na obseg dela. Zavedljiva je tudi navidezna natančnost modela, ki se zrcali v vrednostih konstant, podanih na dve decimalki. Rezultat (E – potrebno delo), čeprav izračunan na dve decimalki natančno, nikakor ne pomeni, da je tudi napoved tako zanesljiva.

6.3.4 Nelinearni modeli

Splošna oblika nelinearnih modelov je:

$$E = (a + bKLOC^c)f(x_1, \ldots, x_n) \tag{6.3}$$

$KLOC$ označuje velikost programa (število programskih vrstic / 1000), E je zopet delo, izraženo v *človek – mesecih*, a, b in c so konstante, $f(x_1, \ldots, x_n)$ pa je korekcija, ki jodoločajo faktorji $x_1 \ldots, x_n$. Osnovno enačbo

$$E = a + bKLOC^c \tag{6.4}$$

izpeljemo s pomočjo regresijske analize iz podatkov preteklih projektov. Osnovni faktor, ki vpliva na obseg dela, je velikost programske opreme, izražena v številu programskih vrstic. Korekcijski faktorji pa bodisi zmanjšajo ali povečajo osnovno vrednost. Če je korekcijski faktor "izkušenost projektne skupine", so vrednosti korekcijskega faktorja lahko na primer 1.50, 1.20, 1.00, 0.80, 0.60 za zelo majhno, majhno, povprečno, visoko in zelo visoko izkušenost.

Tabela 6.3: Trije osnovi nelinearni modeli

Avtorji	Enačba
Halstead	$E = 0.7KLOC^{1.50}$
Boehm	$E = 2.4KLOC^{1.05}$
Walston-Felix	$E = 5.2KLOC^{0.91}$

Vrednosti konstant a, b in c v osnovni formuli se pri različnih avtorjih razlikujejo (tabela 6.3). Velike razlike, ilustrirane v tabeli 6.4, so posledica različnih projektov, ki so služili kot izhodišče za sintezo modelov. Te razlike opozarjajo, da *ne smemo slepo sprejeti kateregakoli modela za ocenjevanje. Vsak model, ki ga želimo uporabljati, moramo umeriti in prilagoditi vrednosti konstant na osnovi naših preteklih projektov.* Eksponent (konstanta c) v nelinearnem modelu odloča, če se se delo za izdelavo enote programske

opreme ($KLOC$ v tem primeru) draži ali ceni, ko se veča obseg dela. Pri večini industrijskih izdelkov se cena na enoto proizvoda manjša z velikostjo serije. Vendar ima le Walston-Felixov model vrednost konstante c manj kot ena. Pri drugih dveh je konstanta c večja od ena, kar pomeni, da se z velikostjo programske opreme veča tudi potrebno delo oziroma cena programske vrstice. To si lahko razložimo s tem, da se z velikostjo programske opreme veča tudi kompleksnost programske opreme.

Tabela 6.4: Vrednosti E za nekaj različnih vrednosti $KLOC$ pri treh osnovnih nelinearnih modelih

KLOC	$E = 0.7KLOC^{1.50}$	$E = 2.4KLOC^{1.05}$	$E = 5.2KLOC^{0.91}$
1	0.7	2.4	5.2
10	22.1	26.9	42.3
100	700.0	302.1	343.6
1000	22135.9	3390.1	2792.6

Model COCOMO

Boehm [6] je svoj model poimenoval COCOMO (COnstructive COst MOdel) in ga določil na treh zahtevnostnih nivojih: osnovnem, srednjem in podrobnem.

1. **Osnovni COCOMO** ima obliko

$$E = bKLOC^c \tag{6.5}$$

Konstanti b in c osnovnega modela COCOMO imata naslednje vrednosti:

Vrsta projekta		
organski	$b = 2.4$	c= 1.05
polpovezan	$b = 3.0$	c= 1.12
vgrajen	$b = 3.6$	c= 1.20

- Organski projekti so tisti, ki jih razvija relativno majhna in izkušena skupina v znanem okolju.
- Vgrajeni projekti so taki, kjer je za programsko opremo veliko omejitev (sistem je vgrajen v zelo nefleksibilno okolje). Primer takega projekta je na primer sistem za nadzor zračnega prometa.

- Polpovezani projekti so vmesna stopnja med organskimi in vgrajenimi.

2. **Srednji COCOMO** v enačbo vpeljuje še 15 korekcijskih faktorjev, ki vplivajo na produktivnost projektne skupine in osnovno oceno povečajo ali zmanjšajo. Vrednosti korekcijskih faktorjev za srednji COCOMO so podani v tabeli 6.5.

Tabela 6.5: Korekcijski faktorji za srednji model COCOMO

	Stopnja					
Korekcijski faktor	nizka	nižja	normalna	višja	visoka	najvišja
Lastnosti produkta						
zanesljivost	0.75	0.88	1.00	1.15	1.40	
velikost podatkovnih baz		0.94	1.00	1.08	1.16	
kompleksnost	0.70	0.85	1.00	1.15	1.30	1.65
Lastnosti računalnika						
časovne omejitve procesiranja			1.00	1.11	1.30	1.66
omejitve centralnega spomina			1.00	1.06	1.21	1.56
nestabilnost platforme		0.87	1.00	1.15	1.30	
hitrost menjave platforme		0.87	1.00	1.07	1.15	
Lastnosti razvijalcev						
sposobnost analitikov	1.46	1.19	1.00	0.86	0.71	
izkušnje z aplikacijo	1.29	1.13	1.00	0.86	0.71	
sposobnost programerjev	1.42	1.17	1.00	0.86	0.70	
izkušnje s platformo	1.21	1.10	1.00	0.90		
izkušnje s programskih jezikom	1.14	1.07	1.00	0.95		
Lastnosti projekta						
uporaba modernih razvojnih metod	1.24	1.10	1.00	0.91	0.82	
uporaba razvojnih orodij	1.24	1.10	1.00	0.91	0.83	
zahtevnost urnika	1.23	1.08	1.00	1.04	1.10	

3. **Podrobni COCOMO** je podoben srednjemu, s tem da uporablja več različnih tabel korekcijskih faktorjev.

6.3.5 Metode s funkcijskimi točkami

DeMarco [10] je predlagal metodo ocenjevanja na osnovi funkcijskih točk. DeMarco uporablja diagrama pretoka podatkov za modeliranje sistema. Na najbolj splošni ravni diagrama pretoka podatkov so prikazane funkcije v uporabniškem vmesniku. Število teh funkcij ali funkcijskih primitivnih operacij (FP - functional primitives) je osnova za točkovanje. Nekatere od teh

funkcij so seveda bolj zahtevne od drugih, zato je potrebno uporabiti nek korekcijski sistem, ki bo upošteval velikost in kompleksnost funkcije.

Vsaka funkcija ustreza neki transformaciji v diagramu pretoka podatkov, velikost transformacije pa je odvisna od števila vhodnih in izhodnih elementov, ki ga označimo z TC. Velikost funkcije je tako:

$$\mathcal{SIZE}(FP) = TC_{FP} \times \log_2(TC_{FP}). \tag{6.6}$$

Kompleksnost se upošteva s korekcijskimi faktorji, s katerimi se množi izhodiščna vrednost. Vsako od funkcij moramo razvrstiti v eno od 10 kategorij (npr. funkcije, ki analizirajo podatke; funkcije, ki opravljajo komleksne izračune itd.). Uteži imajo vrednosti od 0.5 do 2.0. Za celo množico funkcij je vsota točk enaka:

$$FP' = \sum_i \alpha_i \times TC_i \times \log_2(TC_i) \tag{6.7}$$

kjer so α_I korekcijski faktorji vsake funkcije i. Delo (E), potrebno za razvoj programske opreme, ocenjene na tak način, pa je:

$$E = a \times FP'^b \tag{6.8}$$

kjer sta a in b konstanti, določeni na osnovi preteklih projektov.

6.4 Upravljanje s konfiguracijami

V projektni skupini za razvoj programskih projektov so izjemnega pomena dobre komunikacije, od neformalnih (telefon, elektronska pošta) do formalnih (sestanki, delovna knjiga). Izjemnega pomena je tudi upravljanje z vsemi elementi (koda, dokumentacija, zahteve, rezultati testiranja itd.), ki nastanejo tekom celotnega projekta. To imenujemo upravljanje s konfiguracijami (configuration management). V toku projekta ti različni elementi ne le nastajajo, temveč se tudi spreminjajo. Velikokrat hkrati obstaja celo več verzij istih elementov (na primer za različne vrste računalnikov).

Da bi lahko uvedli red in pregled nad to množico elementov, je potrebno v vsakem trenutku vzdrževati tako imenovano *uradno verzijo* vseh elementov. Vsi elementi, ki se uvrstijo v uradno verzijo, morajo biti formalno sprejeti ali dogovorjeni, in služijo kot osnova nadaljnjemu razvojnemu delu.

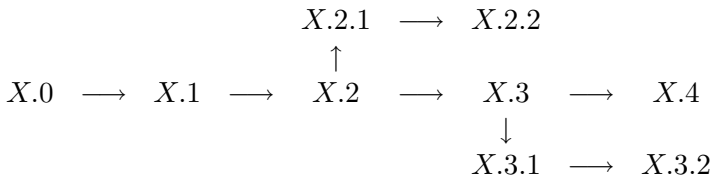
Tipični elementi take uradne verzije so:

- izvorna koda,

- objektna koda,
- zahteve,
- načrti,
- načrt testiranja in testni primeri,
- rezultati testiranja,
- navodila za uporabo

Za shranjevanje uradne verzije obstajajo posebne podatkovne baze, ki ne le skrbijo za hrambo, temveč tudi za pravilen dostop in spreminjanje elementov. Za vsak poseg (vnos, spremembo) v uradno verzijo je potreben formalni zahtevek. Zahtevke je potrebno skrbno preučiti. Če gre za spremembe obstoječih elementov, se mora ugotoviti, kako to vpliva na druge elemente. Včasih je potrebno tudi druge elemente uskladiti s sprejeto spremembo. Če gre za nov element, ga je potrebno prej skrbno testirati. Če je poseg odobren, se mora element med vnosom v bazo "zakleniti", tako da ga drugi ta čas ne morejo niti prebrati.

Osnovno načelo upravljanja s konfiguracijami je, da se vse stare verzije shranjujejo, tako da se je možno vrniti v prejšnje stopnje razvoja. Za označevanje verzij se uporablja decimalni sistem. Tako različne verzije elementa X označujemo X.0, X.1, X.2 itd. Kadar se razvoj razveji, ena veja nadaljuje z X.3, X.4, druga pa z X.2.1, X.2.2 itd. (slika 6.5).



Slika 6.5: Verzije programov označujemo z decimalnim sistemom številčenja

Za izvorno kodo obstaja cela vrsta močnih orodij za podporo (npr. `Make`), ki poleg shranjevanja znajo sestaviti, prevesti in združiti elemente v delujoč program.

V projektni načrt razvoja programske opreme zato sodi tudi načrt za upravljanje konfiguracij, ki naj določi formalne postopke, načine poimenovanja in identifikacije elementov, orodja in metode.

6.5 Kvaliteta programske opreme

V zadnjih nekaj letih je vse več govora o kvaliteti. Podjetja ugotavljajo, da se izplača dobra kvaliteta izdelkov in storitev. Kvalitetno programsko opremo je tudi lažje prodajati, saj imajo kupci z njo manj težav in dodatnih stroškov. Kvaliteta pa je pri razvoju programske opreme nujna tudi zato, da se sploh lahko izdelajo določeni kompleksni sistemi. Brez skrbi za kvaliteto bi določenih velikih in kompleksnih sistemov ne bilo možno zgraditi. Ker igra programska oprema vedno večjo vlogo v človeški družbi, je kvaliteta potrebna tudi za preprečevanje škode in celo izgube življenj. Dokumentiranih je precej primerov, ko je napaka v programski opremi bila neposredni krivec za smrt.

Kot posledica vedno večje zavesti o potrebi po kvaliteti se v svetu pojavljajo novi standardi in predpisi. Nekaj je dokaj splošnih, kot na primer skupina standardov ISO 9000, drugi so zopet namenjeni posebej programski opremi. Vsekakor je v zvezi s kvaliteto programske opreme še veliko nejasnosti.

6.5.1 Kaj je kvaliteta?

Težko je našteti, kaj vse vpliva na kvaliteto programske opreme. Še težje je kvaliteto izmeriti. Presenetljivo pa je, da je pogosto kvaliteto zelo lahko identificirati. Z opredeljevanjem kvalitete programske opreme se ukvarjajo številne sheme in metode. Tako lahko ločimo notranje in zunanje vidike kvalitete, nižje in višje, merljive in le subjektivno določljive. McCall [40] je predlagal obširno shemo za opredeljevanje kvalitete. Na visokem nivoju je določil tri kategorije faktorjev kvalitete, ki so našteti v tabeli 6.6.

Na nižjem nivoju pa je definiral 23 kriterijev kvalitete (npr. popolnost, sledljivost, modularnost, samodokumentiranje, splošnost, preprostost itd.) Faktorji in kriteriji so med seboj povezani tako, da je potrebno za doseganje vsakega faktorja kvalitete izpolnjevati določene kriterije. Za *zanesljivost* na primer, je potrebna *popolnost, konsistentnost in sledljivost*. Težava tega in podobnih sistemov je, da so faktorji kvalitete med seboj odvisni, nekateri celo v negativnem smislu. Če želimo visoko učinkovitost, to slabo vpliva na možnost vzdrževanja, možnost testiranja, fleksibilnost in prenosljivost.

Na kvaliteto lahko gledamo z različnih stališč, ki so vsa relevantna. Garvin [17] razlikuje naslednjih pet definicij kvalitete programske opreme:

1. Transcedentalna definicija govori o notranji kvaliteti, ki jo je težko definirati, izkušeni poznavalci pa jo začutijo.

Tabela 6.6: Tri skupine faktorjev kvalitete programskega produkta

Delovanje produkta	
pravilnost	Ali dela, kar zahtevam?
zanesljivot	Ali deluje ves čas pravilno?
učinkovitost	Ali dela na mojem računalniku kar se da hitro?
varnost	Ali je varen?
uporabnost	Ali ga znam uporabljati?
Revizija produkta	
možnost vzdrževanja	Ali ga lahko popravim?
možnost testiranja	Ali ga lahko testiram?
fleksibilnost	Ali ga lahko spreminjam?
Tranzicija produkta	
prenosljivost	Ali ga lahko uporabljam na drugih računalnikih?
ponovna uporabljivost	Ali lahko ponovno uporabim del programske opreme?
združljivost	Ali ga lahko povežem z drugimi sistemi?

2. Uporabniška definicija govori predvsem o uporabnosti sistema in kako sistem rešuje potrebe uporabnika. Ker se uporabniki med seboj zelo razlikujejo, je nek sistem lahko za nekoga dober, za drugega pa slab. Sistem LaTeX za urejanje besedil je za povprečnega uporabnika preveč zapleten, za znanstvenika pa primeren.

3. Definicija na osnovi produkta zadeva lastnosti programske opreme. Večina sistemov za opredeljevanje kvalitete, tako kot tudi zgoraj opisani, govorijo o tej vrsti kvalitete.

4. Definicija na osnovi specifikacije opredeljuje kvaliteto glede na ujemanje s specifikacijo. To vrsto kvalitete testiramo s sistemskim testom.

5. Vrednostna definicija kvalitete se ukvarja s stroški in dobički pri načrtovanju in vodenju celotnega projekta.

6.5.2 Standardizacija kvalitete

Pri zagotavljanju kvalitete je možno izbrati dve poti:

- preveriti (testirati) kvaliteto gotovega izdelka,

- organizirati tak razvojni oziroma produkcijski proces, da bo izdelek zagotovo ustrezal določenim kriterijem kvalitete.

Najprej je prevladoval prvi način, sedaj pa vse bolj drugi, saj je potratno izdelati nek produkt in ga potem zavreči, če ne ustreza. Pri maloserijskih ali unikatnih produktih pa je to še toliko bolj res.

ISO, mednarodna organizacija za standardizacijo, je izdala številne standarde, ki govorijo o kvaliteti [27]. Standard ISO 9000 daje smernice za izbiro in uporabo serije standardov o kvaliteti. Za razvoj programske opreme je iz te serije najbolj primeren standard ISO 9001 "Modeli kvalitete – model za zagotavljanje kvalitete v načrtovanju, razvoju, izdelavi, inštalaciji in servisiranju". Standard predpisuje kakšne zahteve se mora izpolnjevati v pogodbenem odnosu med naročnikom in razvijalcem. Standard torej zagotavlja kvaliteto po drugi poti iz zgornje delitve.

Ker standardi ISO 9000 niso namenjeni specifično razvoju programske opreme, se pojavljajo še drugi formalni in neformalni standardi, ki so namenjeni posebej programski opremi. Med organizacijami, ki se ukvarjajo s temi posebnimi standardi, je posebej potrebno omeniti organizacijo IEEE (Institute of Electrical and Electronics Engineers).

Če se neka razvojna skupina odloči izboljšati kvaliteto svojih programskih izdelkov, si mora:

1. postaviti določene cilje kvalitete,
2. oceniti svoj način dela in izbrati akcije za izboljšavo tega procesa,
3. med delom zbirati podatke o tem procesu ter o rezultatih oziroma produktih,
4. s pomočjo smiselnih hipotez interpretirati zbrane podatke,
5. če je potrebno, zopet korigirati proces dela.

6.6 Orodja za razvoj programske opreme

Prva programska *orodja za razvoj programske opreme* so bili prevajalniki, urejevalniki in razhroščevalniki. Podpirali so predvsem fazo kodiranja. Ko so se v 50-tih letih pojavili prvi prevajalniki za višje programske jezike, so jih programerji poimenovali kar orodja za avtomatično programiranje, saj so izredno olajšali razvoj novih programov. Pod avtomatskim programiranjem danes razumemo veliko več: razvoj programov neposredno iz formalno podanih zahtev. Postopoma je bilo razvitih vedno več orodij za podporo posameznih faz v razvoju programske opreme, kot so programi za mrežno

planiranje, programi za analizo, za testiranje in podobno. Razvoj programske opreme pa je enovit proces, kjer so posamezne faze med seboj tesno prepletene. Zato se danes aktivno razvija nova orodja CASE[1], ki skrbijo za pomoč pri celotnem razvojnem ciklu nove programske opreme. Orodja CASE skrbijo za konsistentnost vseh faz v razvoju in s svojo enovitostjo vsilijo potrebno metodologijo in sistematizacijo. Prva orodja CASE so predvsem računalniško podprla klasični razvojni cikel, tako da so poenostavila predvsem administrativno delo (pisanje, risanje digramov) in preprečila najbolj grobe napake. Večja avtomatizacija pa je možna le tako, da v orodja CASE vgradijo nekatere od zgoraj omenjenih metodologij, kot so uporaba formalnih jezikov za specifikacijo in logično dokazovanje pravilnosti programov.

6.6.1 Splošni osnutek za projektni načrt

Za konec poglavja si oglejmo še osnutek za projektni načrt razvoja programske opreme. Našteti so predvsem elementi, ki v načrtu ne smejo manjkati.

1. Okvirni podatki o projektu:
   (a) namen,
   (b) glavne funkcije,
   (c) sistemske specifikacije (število uporabnikov, zahtevana hitrost odziva, zanesljivost itd.),
   (d) razvojni scenarij.
2. Potrebne zmogljivosti:
   (a) ljudje (izobrazba, izkušnje, čas),
   (b) strojna oprema za razvoj (opis, dosegljivost, dobavni rok),
   (c) razvojna programska oprema (urejevalniki besedil, prevajalniki, razhroščevalniki, orodja CASE).
3. Načrt:
   (a) razdelitev na aktivnosti,
   (b) mrežni načrt z vsemi diagrami in tabelami.

[1] Computer Aided System Engineering

Poglavje 7

Analiza zahtev

Analiza zahtev je prvi korak k uspešni izpeljavi projekta razvoja programske opreme. V tej fazi je potrebno do podrobnosti identificirati in dokumentirati zahteve bodočega uporabnika oziroma naročnika sistema. Te zahteve niso le dejanske funkcije, ki jih mora programska oprema podpirati, temveč cela vrsta dodatnih zahtev, kot so zmogljivost, zanesljivost, uporabnost in cena sistema. Med analizo nas podrobno še ne zanima, kako bodo te zahteve tudi uresničene, saj je to naloga naslednje faze načrtovanja.

Zahteve, ki jih odkrijemo med analizo, je potrebno sistematično podati v posebnem dokumentu—**specifikaciji zahtev**. Specifikacija mora odsevati tako zahteve naročnika kot spoznanja analitika. Specifikacija je osnova za nadaljnje delo in na koncu tudi kriterij za vrednotenje uspeha celotnega projekta, to je tega, v kolikšni meri so se zahteve uresničile.

Specifikacija zahtev je osnova za naslednjo fazo v razvoju, to je načrtovanje. Pri načrtovanju je potrebno izdelati načrt programske opreme; iz kakšnih sestavnih delov (modulov) naj bo zgrajena in kako so ti sestavni deli med seboj povezani (vmesniki in sistemska arhitektura). Faza načrtovanja se tako zaključi z načrtom programske opreme.

Analize zahtev in načrtovanja programske opreme pa ni možno vedno popolnoma ločiti. Analiza pogosto vsebuje že elemente visokonivojskega načrtovanja sistema. Analizi lahko tudi sledi preliminarno načrtovanje, ki ima za posledico spremembo ali dopolnitev specifikacije. Takšna postopna izboljšava specifikacije je bistvo razvoja s pomočjo prototipov. Drugi razlog, zakaj sta analiza zahtev in načrtovanje tesno povezana ali celo prepletena, pa je uporaba določene metodologije ali notacije. Uporaba metod podatkovnega toka ali podatkovne strukture, objektno orientirani pristop oziroma izbrano orodje CASE zahtevajo usklajeno rabo v obeh fazah razvoja

programske opreme.

Analiza zahtev poteka po naslednjih osnovnih korakih:

1. Seznanjanje s problemom—študij projektnega načrta.

2. Ocenjevanje strukture in povezave informacijskega pretoka na problemskem področju in ugotavljanje glavnih omejitev. Na osnovi tega je potrebno oblikovati eno ali več rešitev, ki izpolnjujejo zahteve, in če je potrebno, izdelati prototip.

3. Izdelava specifikacije.

4. Recenzija rezultatov analize skupaj z uporabnikom.

5. Če je potrebno, ponoviti točke 2–4, dokler uporabnik ni zadovoljen s predlagano rešitvijo oziroma po potrebi spremeniti in dopolniti projektni načrt.

Zahteve je smiselno razvrstiti na več skupin po stopnji pomembnosti[1]. Uporabniki imajo namreč često težave z izražanjem svojih zahtev. Zato se lahko zgodi, da se veliko truda porabi za malo pomembne funkcije. Z razvrščanjem po pomembnosti pa se temu izognemo in v implementaciji poskrbimo najprej za najpomembnejše zahteve.

Poleg dejanskih funkcij bodočega sistema se je pri analizi potrebno vprašati, na kakšni strojni opremi naj bi se sistem izvajal, s pomočjo katerega operacijskega sistema in koliko delovnih mest oziroma terminalov naj bi podpiral. Pomembno je, koliko in kakšne skupine uporabnikov obstajajo, kakšno predznanje imajo in kakšna naj bo njihova dostopnost do vseh funkcij sistema. Kadar gre za sisteme s podatkovnimi zbirkami, je pomembno, kako obsežne so, saj od tega zavisi na primer implementacija algoritmov za iskanje in urejanje. Pomemben je odzivni čas sistema in navsezadnje cena razvoja sistema.

Poleg samih lastnosti sistema je pomembno, kako se sistem povezuje s svojim okoljem. Zato je potrebno analizirati tako neposredno okolje, v katerem bo sistem deloval kot način interakcije sistema z okoljem. Pri vpeljavi novih informacijskih sistemov so običajno potrebne kadrovske spremembe, na primer šolanje in premeščanje ljudi ali celo zaposlovanje novih ljudi z drugačno izobrazbo. Sistemi, ki zahtevajo velike spremembe v načinu dela in mišljenju ljudi, lahko propadejo prav zato, ker razvijalci premalo upoštevajo okolje, v katerem naj bi sistem deloval.

[1] Dovolj sta že dve skupini zahtev, ki jih v angleščini imenujemo “MUSTs” in “WANTs”.

Zaradi vseh naštetih vidikov je vsaj pri večjih projektih smiselno izdelati posebno študijo izvedljivosti projekta (angl. feasability study). Tehnična študija izvedljivosti mora odgovoriti, če predvidena strojna oprema zadostuje potrebam in če je možno enostavno uvesti sistem v novo okolje. Včasih je priprava podatkov v obliki, primerni za novi sistem, bolj zahtevna in zamudna kot sam razvoj sistema. Ekonomska študija izvedljivosti pa mora odgovoriti, ali koristi novega sistema odtehtajo stroške. Čeprav je težko natančno ocenjevati stroške razvoja programske opreme, je še težje izraziti direktne in posredne prihranke.

Pri sistemih, ki se morajo vklopiti v obsežnejši organizacijski sestav, pa je še posebej pomembno, da upoštevajo rezultate splošne analize informacijskih zahtev, da bi se novi sistem lahko brez težav povezoval z drugimi sistemi.

7.1 Specifikacija zahtev

Specifikacija zahtev je dokument, v katerem so zbrani rezultati analize. Te rezultate mora posredovati vsem vpletenim, tako razvijalcem kot naročnikom. Za razvijalce je specifikacija izhodišče za načrtovanje, zato mora biti kar se da natančna, zaželen je formalni ali matematični zapis. Za naročnike pa mora biti tudi razumljiva, kar pomeni, da mora biti berljiva, to pomeni napisana v naravnem jeziku in opremljena s slikami.

IEEE [26] poleg tega priporoča, da je specifikacija:

Enoumna — zahteve je možno le enolično interpretirati. Zaradi svoje narave je to z naravnim jezikom težko.

Popolna — vse pomembne zadeve v zvezi s funkcionalnostjo, zmogljivostjo in omejitvami naj bodo dokumentirane. Včasih nekaterih zahtev v fazi analize res še ni možno dokončno oblikovati. V takih primerih pa mora specifikacija vsaj določiti, kdaj naj se to zgodi.

Preverljiva — pomeni, da je možno ugotoviti, če so zahteve uresničene. Fraze, kot so "sistem naj bo prijazen do uporabnika" niso objektivno preverljive.

Konsistentna — zahteve si ne smejo nasprotovati niti v logičnem niti v časovnem smislu.

Spremenljiva — ker programska oprema modelira neko realnost, se mora skupaj s to realnostjo spreminjati. Zato se mora spreminjati tudi njena specifikacija. Specifikacija zahtev mora biti tako urejena, da je vanjo zlahka možno vnašati spremembe.

Sledljiva — vzrok za vsako zahtevo mora biti natanko znan.

Uporabna — specifikacija mora biti uporabna tudi, ko sistem že deluje. Implicitno védenje, ki je igralo vlogo pri pisanju specifikacije, pa ni bilo zapisano, lahko povzroča velike težave pri vzdrževanju programske opreme.

ANSI/IEEE standard [26] priporoča naslednjo strukturo specifikacije:

1. Uvod:
   (a) namen,
   (b) obseg,
   (c) definicije, kratice in okrajšave,
   (d) reference,
   (e) pregled produkta.
2. Splošni opis:
   (a) obeti produkta,
   (b) funkcije produkta,
   (c) karakteristike uporabnikov,
   (d) splošne omejitve,
   (e) predpostavke in odvisnosti.
3. Specifične zahteve:
   (a) Funkcijske zahteve:
      i. 1. funkcijska zahteva:
         A. uvod,
         B. vhodni podatki,
         C. procesiranje,
         D. izhodni podatki.
      ii. 2. funkcijska zahteva:
      iii. ...
   (b) Zunanji vmesniki:
      i. uporabniški vmesniki,
      ii. strojni vmesniki,

iii. programski vmesniki,
iv. komunikacijski vmesniki.

(c) Zmogljivost

(d) Omejitve pri načrtovanju:
i. skladnost s standardi,
ii. omejitve strojne opreme.

(e) Atributi:
i. varnost,
ii. zmožnost vzdrževanja,
iii. ...

(f) Ostale zahteve:
i. podatkovne baze,
ii. upravljanje s produktom,
iii. prilagoditve na lokacijo,
iv. ...

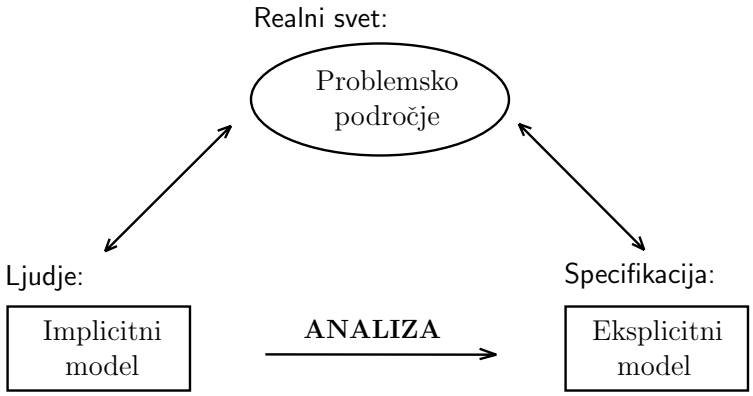
Uspeh projekta razvoja programske opreme je v veliki meri odvisen od pravilne specifikacije. Majhne napake v specifikaciji lahko zahtevajo velike spremembe v kodi, saj programska oprema ni zvezna.

7.2 Ljudje kot viri informacij

Naloga analitikov, strokovnjakov za analizo, je iz slabo definiranih, nejasnih, včasih celo nasprotujočih si informacij o nekem realnem problemu izluščiti jasne in eksplicitne zahteve. Pri določanju zahtev in študiju uporabnikovega okolja se pogosto pojavi težava, da na uporabnikovi strani ni kompetentnega sogovornika.

Med analizo se modelira del neke stvarnosti (slika 7.1). Ta del stvarnosti imenujemo problemsko področje (angl. universe of discourse). Primeri takih problemskih področij so na primer, knjižnični informacijski sistem, proizvodna linija, kontrolni sistem dvigala itd. Naloga analitikov je, da za določeno problemsko področje zgradijo eksplicitni model, ki naj vsebuje vse zadostne in potrebne informacije o tem področju in ki je razumljiv vsem vpletenim ljudem. Eden od glavnih problemov analize je prav v ločevanju pomembnih informacij od nepomembnih podrobnosti.

Ljudje, ki so vpleteni v neko problemsko področje, imajo o njem svoj implicitni model. Implicitni model problemskega področja vsebuje vse možne



Slika 7.1: Naloga analize je, da v specifikaciji določi ekspliciten model dela stvarnosti, ki ga programska oprema modelira.

informacije in znanja o tem področju, ki ga imajo ljudje v veliki meri za samoumevnega. Zato je implicitni model težko ubesediti, saj vsebuje navade, običaje, predsodke, celo nasprotujoča si dejstva. Naloga analitika je torej spremeniti implicitni model v eksplicitnega. Pri tem pa naleti na dve vrsti problemov: probleme analize in probleme pri pogajanjih.

7.2.1 Problemi analize

Problemi se pri analizi pojavijo zato, ker implicitni modeli niso ubesedeni, ker se s časom spreminjajo, ker naročniki in sistemski analitiki ne govorijo istega "jezika" in ker se včasih enostavno ne da vsega implicitnega znanja formalno zapisati.

Pri analizi naj bi naročnik pravilno in popolno opisal problem, ki bi ga rad rešil s programskim produktom. To pa je v praksi težko dosegljivo in ponavadi vodi do nepopolnih ali celo napačnih modelov problemskega področja. Človeški spomin je namreč omejen. Še posebej je omejen njegov kratkoročni spomin, s pomočjo katerega procesiramo informacije. V kratkoročnem spominu naj bi lahko hkrati hranili le okoli sedem informacijskih enot [41]. Dolgoročni spomin ima seveda veliko večjo kapaciteto, vendar je dostop vanj le indirekten, skozi kratkoročni spomin. Med procesiranjem informacij si zato običajno pomagamo še z zunanjimi mediji zapisa — tablo ali listom papirja. Eksplicitni modeli, do katerih smo prišli le s spraševanjem uporabnikov, pa so lahko nepopolni tudi zato, ker ljudje v odgovorih dajo večjo težo trenutni situaciji kot tisti v preteklosti. Vplivajo pa še drugi faktorji, kot so izobrazba, izkušnje itd.

Naročnik ne more jasno izraziti svojih zahtev takrat, ko je pobuda za avtomatizacijo nekega procesa izraz nezadovoljstva s trenutnim stanjem. Avtomatizirati trenutno situacijo zato v takem primeru ni primerna rešitev. Naloga analitikov je, da ugotovijo, katere so prave zahteve in kako se bodo razvijale v prihodnje. Kar polovica vsega vzdrževanja programske opreme ima opravka s prilagajanjem opreme novim in spremenjenim zahtevam.

Pri zbiranju informacij za izdelavo specifikacije zahtev lahko uporabimo naslednje štiri strategije:

1. **Spraševanje:** Naročnike sprašujemo, kaj pričakujejo od novega sistema. Pri tem lahko le upamo, da zna naročnik obiti svoje omejitve in predsodke. Spraševanje lahko poteka v obliki intervjuja, "brainstorminga" ali vprašalnika.

2. **Izpeljava iz obstoječega sistema:** Analizo začnemo na osnovi obstoječega sistema, na primer podobnega sistema v neki drugi organizaciji ali opisa sistema v literaturi.

3. **Sinteza iz lastnosti okolice:** Programska oprema se bo uporabljala v določenem okolju. Da bi uspešno delovala, mora upoštevati zakonitosti okolja. Zahteve lahko zato formuliramo na osnovi analize okolja. Tovrstni analizi pravimo procesna analiza, normativna analiza ali decizijska analiza.

4. **Izdelava prototipov:** Skozi nekaj generacij prototipov, ki vodijo do vedno bolj podrobnih zahtev, pridemo do končne specifikacije. Namesto dejanskih prototipov lahko specifikacijo določimo z razvijanjem scenarija možne uporabe sistema.

Katera strategija zbiranja informacij je primerna, je odvisno od izkušenj razvijalcev in naročnikov, kompleksnosti produkta ter stabilnosti in seznanjenosti s problemskim področjem.

7.2.2 Pogajalski problemi

Večina strategij in analitičnih metod je v svojem bistvu tehnicističnih. Naloge oziroma probleme naj bi rešili z rekurzivnim deljenjem na manjše in preprostejše probleme, od katerih naj bi bil vsak rešljiv z enim najboljšim možnim načinom, ki ga lahko odkrijemo s skrbnim opazovanjem in eksperimentiranjem. Na področju razvoja programske opreme to pomeni, da je dovolj, da analitiki skrbno izprašajo poznavalce problemskega področja in opazujejo uporabnike pri delu. Na ta način naj bi odkrili "dejanske" zahteve.

Med nadaljnjim razvojem zato komunikacija z naročniki ni potrebna. Takšne tehnicistične metode slonijo na predpostavki, da objektivna resnica dejansko obstaja in jo je le potrebno odkriti.

Tehnicističen pristop je primeren za čisto tehnične domene, takrat, ko so vmešani interesi ljudi, pa se običajno ne obnese. Ljudje imamo o svojem okolju nepopolne, subjektivne, iracionalne in nasprotujoče si poglede. Takrat analitik ne more biti le pasiven zunanji opazovalec, temveč mora sam aktivno sodelovati pri oblikovanju modela problemskega področja. Sicer pa tudi pri povsem tehnicističnem pristopu analitik ne more obiti svojih predpostavk o naravi problemskega področja. Množico predpostavk o znanju in načinu pridobivanja znanja iz fizičnega in socialnega sveta imenujemo *paradigma*. Predpostavke ločimo na tiste o načinu pridobivanja znanja (epistemološke predpostavke) in tiste o pogledu na fizični in socialni svet (ontološke predpostavke).

Objektivistični pristop k analizi uporabi naravoslovne metode v socialnem kontekstu. Subjektivističen pristop pa poskuša razumeti stvari skozi oči posameznika in družbe.

Če predpostavke o znanju razvrstimo po subjektivno-objektivni dimenziji, predpostavke o svetu pa po dimenziji red-konflikt, dobimo štiri ekstremne paradigme [24]:

1. **Funkcionalizem** (objektivizem – red). Razvijalec sistema išče merljive vzročne povezave. Učinkovitost in uporabnost je možno objektivno testirati kot v drugih inženirskih disciplinah.

2. **Socialni relativizem** (subjektivizem – red). Analitik omogoča spremembe in spodbuja učenje vseh vpletenih ljudi.

3. **Radikalni strukturalizem** (objektivizem – konflikt). Razvijalci sistema posredujejo v konfliktu med socialnimi razredi. Informacijski sistemi običajno podpirajo interese lastnikov kapitala na račun interesov delojemalcev. Da se vzpostavi ravnovesje, naj bi razvijalci v konfliktu podprli delojemalce. Ta pristop k razvoju naj bi izboljšal delovne razmere.

4. **Neohumanizem** (subjektivizem – konflikt). Glavna skrb je emancipacija vseh sodelujočih. Analitik naj bi deloval kot neke vrste socialni terapevt, ki naj bi odpravil vse motnje in ovire za racionalne odnose.

Naštete štiri paradigme so ekstremni primeri. V praksi običajno uporabimo mešanico različnih predpostavk. Kadar gre za tehnične probleme, kot je na primer avtomatizacija dvigala, je funkcionalni pristop optimalen. Kadar pa

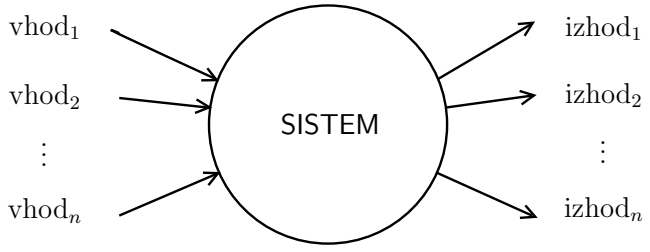
mora sistem podpirati delo ljudi, kot pri avtomatizaciji pisarniškega dela, ni možno obiti njihovega odnosa do sistema. Nezadovoljni uporabniki sistema enostavno ne bodo uporabljali ali ga bodo uporabljali tako, da ne bo učinkovit.

7.3 Orodja za dokumentiranje zahtev

Specifikacija zahtev je dokument, ki mora zadovoljiti dve skupini ljudi, naročnike in razvijalce. Za naročnike ali uporabnike je pomemben jasen in točen opis funkcij, ki jih mora sistem nuditi. Za razvijalce pa je specifikacija vodilo za načrtovanje sistema. Očitno tadva vidika ni lahko združiti. Za uporabnike je specifikacija najbolj razumljiva, če je napisana v naravnem jeziku. Žal so opisi v naravnem jeziku problematični; lahko so redundantni, zamolčijo določene vidike, lahko so pretirano določljivi, kontradiktorni in večpomenski. Za razvijalce je zato primeren bolj formaliziran zapis, ki zagotavlja konsistentnost in bolj popoln opis zahtev.

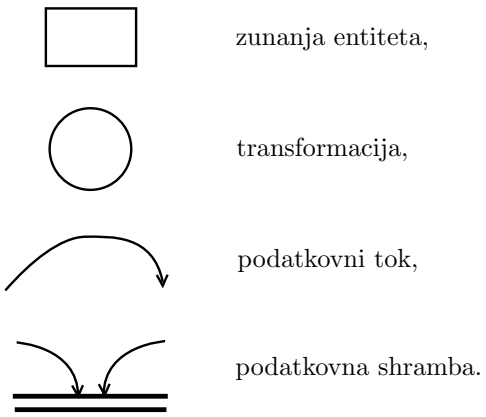
Za namene analize in tudi kasnejšega načrtovanja je bila razvita cela vrsta metod, ki jih sestavlja množica postopkov za ocenjevanje in modeliranje, in ustrezna, običajno grafična notacija. Večino metod je možno uvrstiti v dve široki skupini: metode na osnovi podatkovnega toka in metode na osnovi strukture podatkov.

7.3.1 Analiza na osnovi podatkovnega toka



Slika 7.2: Diagram podatkovnega toka na nivoju celotnega sistema

Informacije, ki tečejo skozi računalniški sistem, se transformirajo. Sistem sprejme na vhodu raznovrstne podatke in jih s pomočjo strojne opreme, programske opreme in uporabnikovega posredovanja spremeni v izhodne podatke, ki imajo zopet lahko zelo različno obliko (slika 7.2). Tak pretok informacij skozi sistem lahko ponazorimo z diagramom podatkovnega toka.



Slika 7.3: Simboli za označevanje diagramov podatkovnega toka

Diagram podatkovnega toka (angl. data flow diagram ali bubble chart) ponazarja pretok podatkov in njihovo transformacijo na tej poti. V diagramih podatkovnega toka uporabljamo simbole prikazane na sliki 7.3.

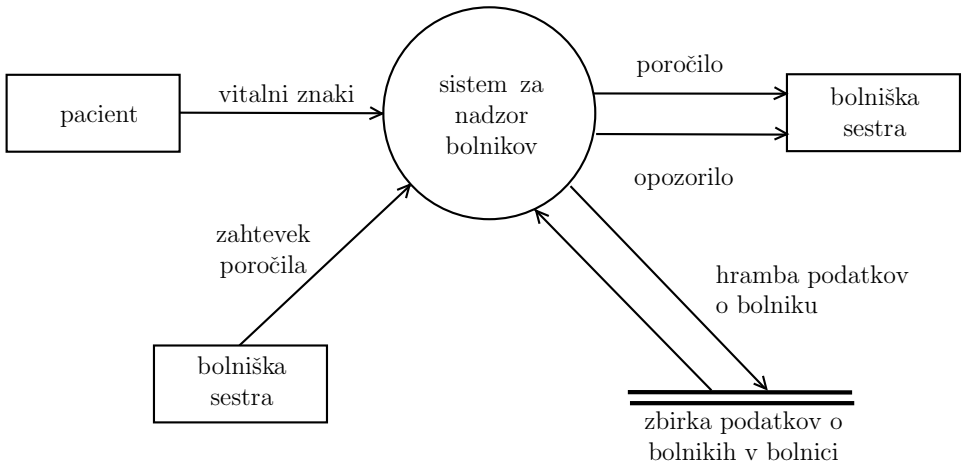
Zunanja entiteta predstavlja izvir ali ponor vhodnih oziroma izhodnih informacij. Transformacije pretvorijo svoje vhodne podatke v izhodne podatke. Podatkovni tok povezuje transformacije med seboj in z zunanjimi entitetami. Puščica označuje smer podatkovnega toka, podatkovna shramba pa shranjuje podatke. Celoten sistem na najvišji ravni (stopnja 01) ponazorimo le z eno transformacijo, ki jo označimo s krogom ali mehurčkom (slika 7.2). S puščicami so označene vhodne in izhodne informacije.

Diagram podatkovnega toka lahko prikazuje različne ravni abstrakcije. Običajno se analiza izvaja tako, da se podrobnosti razkrivajo postopoma. Kot primer si oglejmo analizo sistema za nadzor bolnikov v oddelkih za intenzivno nego.

Stopnja 01 diagrama podatkovnega toka (slika 7.4) prikazuje celoten sistem le z enim samim mehurčkom oziroma transformacijo. Na bolj podrobnih ravneh nato posamezno transformacijo razdelimo, da bi lahko bolj podrobno prikazali tok informacij.

Na stopnji 02 je sistem prikazan s štirimi transformacijami in bolj podrobnim potekom podatkov (slika 7.5).

Na stopnji 03 je še bolj podrobno razdelana transformacija "osrednji nadzor" s stopnje 02 (slika 7.6). Pri izdelavi diagramov podatkovnega toka se držimo načela, da pri prehodu na nižjo raven razdelimo naenkrat le eno od transformacij. Čeprav gre v našem primeru za dokaj preprost sistem, so jasno razvidne poti podatkov v sistemu. Vsako transformacijo bi lahko



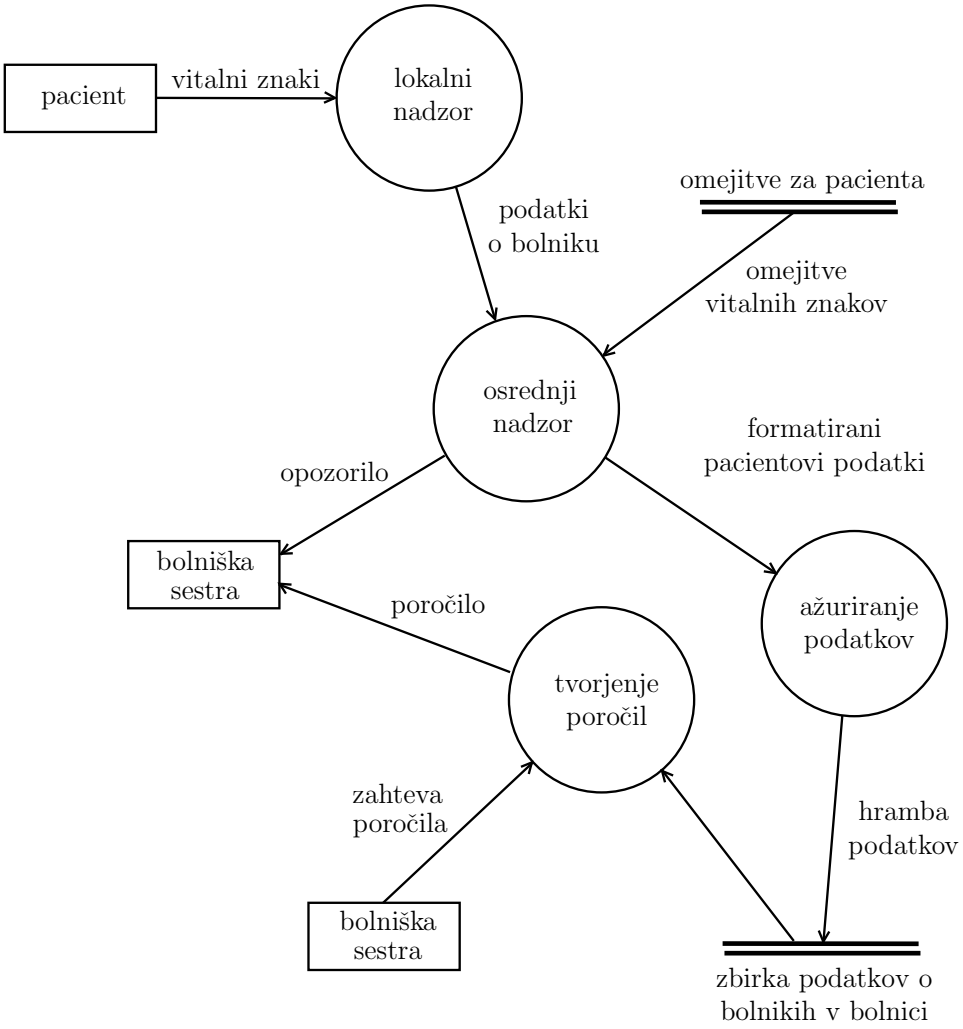
Slika 7.4: Stopnja 01 podatkovnega toka za sistem nadzora bolnikov

še bolj podrobno analizirali, da bi prikazali poljubno raven podrobnosti. Z deljenjem transformacij nadaljujemo tako dolgo, da pridemo do enot, ki ustrezajo posameznim programskim enotam ali modulom. V diagramu pretoka podatkov ni eksplicitno podan niti vrstni red operacij niti kontrolni pogoji, lahko pa to razberemo iz samega diagrama.

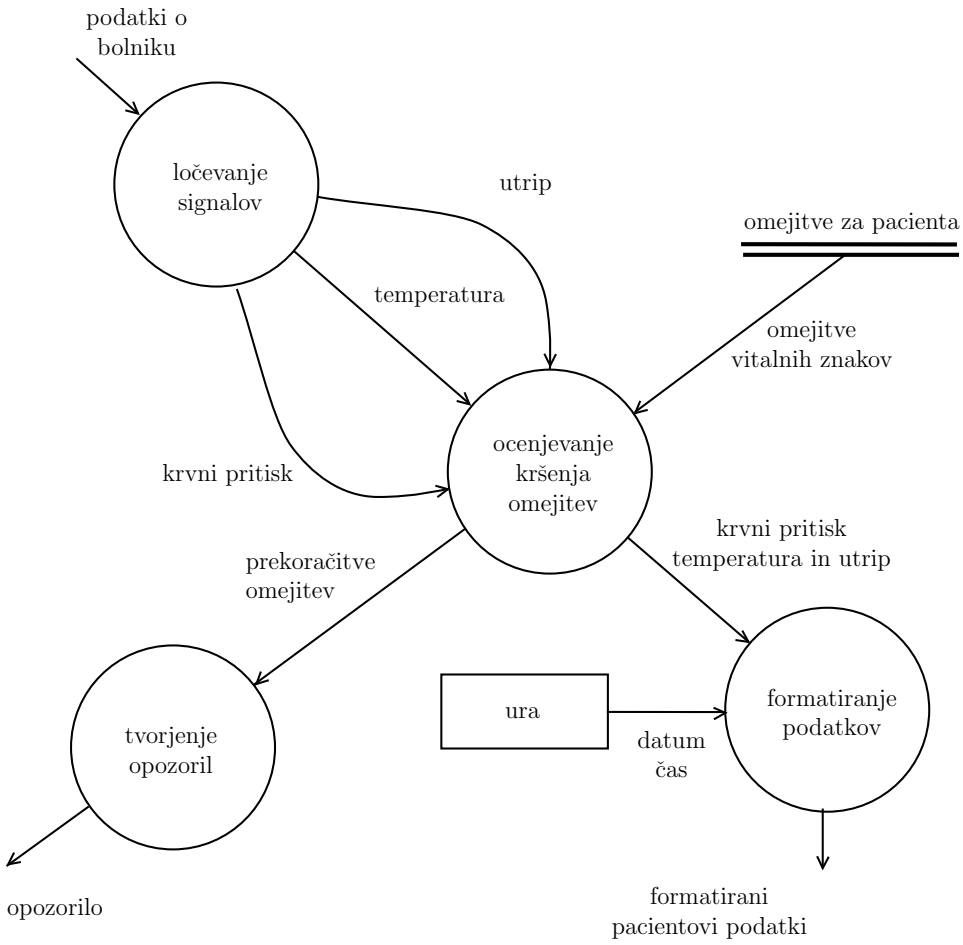
Vsaka puščica v diagramu podatkovnega toka predstavlja neko informacijo. Strukturo teh informacij lahko formalno predstavimo z regularnimi izrazi, ki uporabljajo naslednje konstrukte: zaporedje (+), izbiro [|] in ponavljanje { }^{n}). Primer strukturiranega zapisa slovenskih avtomobilskih registrskih tablic s pomočjo regularnih izrazov je prikazan v tabeli 7.1. Tak strukturiran opis informacij moramo zbrati v *podatkovnem slovarju*. Transformacije pa bolj podrobno opišemo kar v naravnem jeziku ali v strukturirani psevdokodi.

Metoda SADT

Metoda SADT (Structured Analysis and Design Technique) [38] obsega posebno grafično notacijo, metodologijo analize in kasnejše razvito CASE orodje. S pomočjo metode SADT analitik razvije hierarhičen model, ki vsebuje veliko število diagramov, ki prikazujejo medsebojni odnos informacij in funkcij v programski opremi. Osnovna celica diagramov SADT je prikazana na sliki 7.7. Vsako celico lahko hierarhično razgradimo na več celic, podobno kot smo to storili pri diagramih pretoka podatkov.



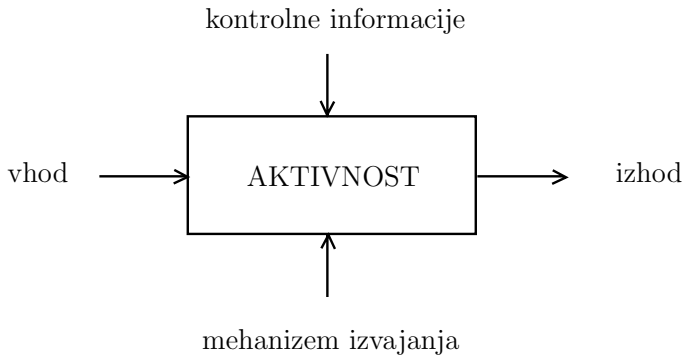
Slika 7.5: Stopnja 02 podatkovnega toka za sistem nadzora bolnikov



Slika 7.6: Podatkovni tok v osrednjem nadzoru (stopnja 03)

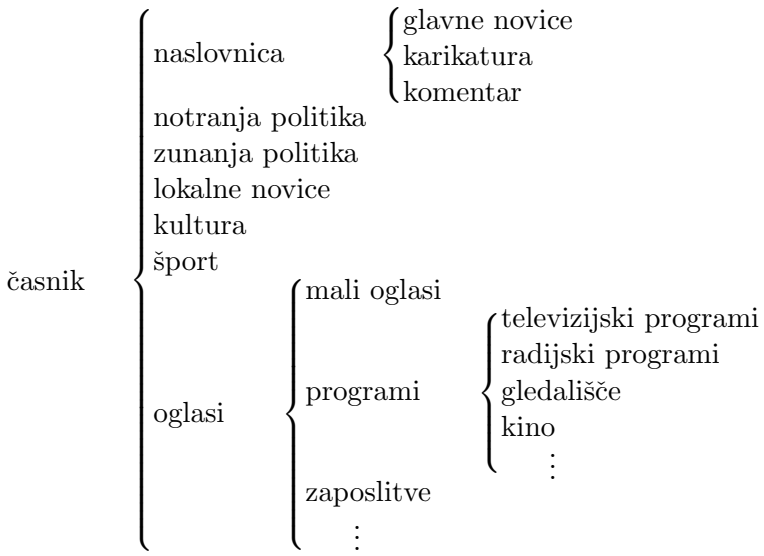
Tabela 7.1: Struktura običajnih (belih) slovenskih avtomobilskih registrskih številk, definirana s pomočjo regularnih izrazov. Dejansko so registrske oznake še bolj omejene, saj so črke v običajnih registracijah le na prvem ali zadnjem mestu. Nekatere kombinacije tudi niso dovoljene. Zaradi lažje ločljivosti v registracijah po naročilu ne sme hkrati nastopati črka "O" in ničla "0".

$$
\begin{aligned}
\text{registrska oznaka} &= \text{krajevna oznaka} + \text{grb} + \text{številka} \\
\text{krajevna oznaka} &= [LJ|MB|CE|KP|KR|NM|MS|KK|SG] \\
\text{številka} &= [\text{številka po naročilu}|\text{običajna številka}] \\
\text{številka po naročilu} &= \{\text{znak}\}^{[3|4|5|6]} \\
\text{znak} &= [\text{črka}|\text{cifra}] \\
\text{običajna številka} &= \{\text{znak}\}^2 + \{-\}^1 + \{\text{znak}\}^3 \\
\text{črka} &= [A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|R|S|T|U|V|Z] \\
\text{cifra} &= [1|2|3|4|5|6|7|8|9|0]
\end{aligned}
$$



Slika 7.7: Osnovni gradnik diagramov SADT

7.3.2 Analiza na osnovi strukture podatkov



Slika 7.8: Struktura dnevnega časnika, prikazana z Warnierovim diagramom

Že pri analizi na osnovi toka podatkov smo opozorili, da je potrebno poleg podatkovnega toka izpostaviti tudi vsebino in strukturo podatkov. Kadar se želimo pri analizi osredotočiti bolj na strukturo podatkov, lahko uporabimo eno od metod, ki temeljijo na strukturi podatkov. Čeprav se metode, ki temeljijo na analizi strukture podatkov, med seboj razlikujejo, je za vse značilno, da je potrebno:

1. identificirati glavne informacijske objekte (entitete ali enote) in operacije (akcije ali procese),
2. informacije hierarhično predstaviti,
3. strukturo informacij predstaviti z regularnimi izrazi,
4. hierarhično podatkovno strukturo na koncu preslikati v programsko strukturo.

Za hierarhično predstavitev strukture podatkov se uporabljajo Warnierovi diagrami [69]. Na sliki 7.8 je s pomočjo Warnierovega diagrama predstavljena struktura dnevnega časnika. Orr [46] je na osnovi Warnierjevega dela razvil metodo DSSD (Data Structured System Development). Metoda DSSD mora najprej prikazati, kako se informacije prenašajo med producenti

in uporabniki informacij z vidika posameznega producenta ali uporabnika informacij. Funkcije bodoče programske opreme ponazorimo z Warnierovo notacijo, kjer je poleg podatkov prikazano tudi njihovo procesiranje. Nazadnje metoda DSSD zahteva še ponazoritev končnih rezultatov z Warnierovim diagramom.

Med znanimi metodami analize na osnovi strukture podatkov sta tudi metoda JSP (Jackson Structured Programming) in metoda JSD (Jackson System Development) [28].

7.3.3 Ostale metode analize

V literaturi objavljeno in na tržišču dosegljivo je veliko število metod oziroma programskih orodij CASE (Computer Aided Software Engineering). Nekatere bolj znane metode oziroma orodja za analizo so:

- metoda SREM (Software Requirements Engineering Methodology) [3],
- metoda PSL/PSA (Problem Statement Language/Problem Statement Analyzer) [64],
- metoda TAGS (Technology for the Automated Generation of Systems) [58],
- metoda SSADM (Structured Systems Analysis and Design Method) [12].

Objektno orientirane metode analize bomo obravnavali skupaj z objektno orientiranim načrtovanjem v 8. poglavju.

Izbira metode je odvisna predvsem od narave problema. Če informacija v sistemu, ki naj bi ga avtomatizirali, pogosto spremeni svojo obliko, potem je primerna analiza podatkovnega pretoka. Pri močno strukturiranih podatkih pa je primernejša analiza na osnovi strukture podatkov.

7.4 Verifikacija in validacija

Za uspešen zaključek razvoja programske opreme moramo po vsaki fazi v razvoju preverjati njeno uspešnost in pravilnost. Za analizo je potrebno podrobno oceniti, ali so vsi aspekti sistema dovolj natančno opisani: pravilnost, popolnost, konsistentnost, natančnost, berljivost in zmožnost testiranja. Preveriti moramo vse vmesnike in oceniti zahteve v luči vseh možnih scenarijev za uporabo sistema.

Poleg preverjanja rezultatov analize je to tudi primeren trenutek za snovanje načrta testiranja v naslednjih fazah razvoja. Načrt testiranja predpisuje, katere elemente je potrebno testirati in kdaj jih je potrebno testirati. Predvsem je pomembno načrtovanje sistemskega testiranja, ko dokončan sistem primerjamo s specifikacijo zahtev.

7.5 Zaključek

Glavna načela, ki se jih držimo pri analizi, so:

- Problem moramo razdeliti na podprobleme tako, da se podrobnosti razkrijejo postopoma in na hierarhičen način.
- Specifikacija mora obsegati celoten sistem, katerega del je programska oprema, in opisati vse okoliščine, v katerih sistem deluje.
- Ločiti moramo funkcije od njihove implementacije. Odgovoriti moramo na vprašanje, **KAJ** je potrebno, preden odgovorimo na vprašanje **KAKO** to narediti. Specifikacija je model za *razumevanje*, ne pa *načrt* za implementacijo.
- Na osnovi specifikacije naj bo moč ugotoviti, če nek sistem izpolnjuje zahteve.
- Specifikacija naj bo modularna. Če pride do sprememb, naj bo potrebno spremeniti čim manjši del specifikacije.

Poglavje 8

Načrtovanje

Cilj faze načrtovanja programske opreme je izdelava načrta programske opreme. Osnova za načrtovanje je specifikacija zahtev, kjer so opisane vse zahtevane funkcije programske opreme ter morebitne druge zahteve, kot so na primer potrebna strojna in programska oprema, prilagojenost standardom ali zahteve po dodatnem izobraževanju uporabnikov. Načrt programske opreme pa je nato izhodišče za kodiranje, ki je naslednja faza v razvoju programske opreme.

Sistematičen pristop k načrtovanju programske opreme je izrednega pomena za uspeh celotnega razvoja. Kvaliteta programske opreme je v največji meri odvisna prav od dobrega načrtovanja. Že na osnovi načrta programske opreme je možno oceniti kvaliteto bodoče programske opreme. Le s pomočjo načrtovanja lahko uporabnikove zahteve uresničimo v programskem izdelku, saj je načrt osnova za vso nadaljnjo izdelavo. V preteklosti, včasih pa še sedaj, če se zelo mudi, so razvijalci začeli izdelavo programske opreme kar s pisanjem kode, brez načrta. Rezultat takega nenačrtnega razvoja je nestabilna programska oprema, ki odpove že pri majhnih spremembah. Slabo načrtovano ali programsko opremo, ki je nastala brez načrta, je tudi težko testirati in vzdrževati. Na dolgi rok je taka nekvalitetna oprema veliko dražja od skrbno načrtovane.

Načrtovanje programske opreme se začne z opisom programske opreme na visokem nivoju, ki ga postopoma dopolnjujemo in širimo, da postane opis dovolj podroben za kodiranje. Načrt ima zato hierarhično in modularno strukturo. Z vidika vodenja zato ločimo vsaj dve razdobji v tej fazi; preliminarno in podrobno načrtovanje. S tehničnega vidika pa ima načrt tri sestavine:

1. načrt programske arhitekture,

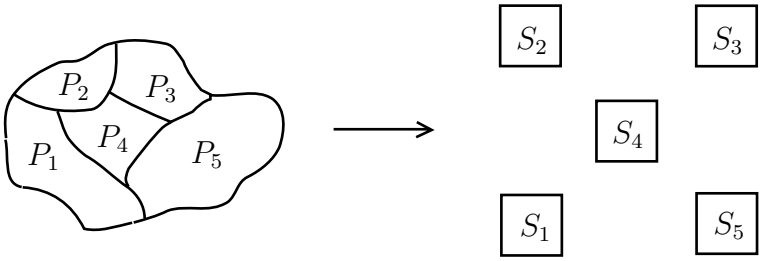
2. načrt podatkovnih struktur,

3. načrt posameznih procedur (modulov).

V nadaljevanju poglavja si oglejmo zato najprej, kaj naj obsega načrt programske opreme, katera načrtovalska načela naj uporabljamo in katere metode načrtovanja poznamo.

8.1 Sestavine načrta programske opreme

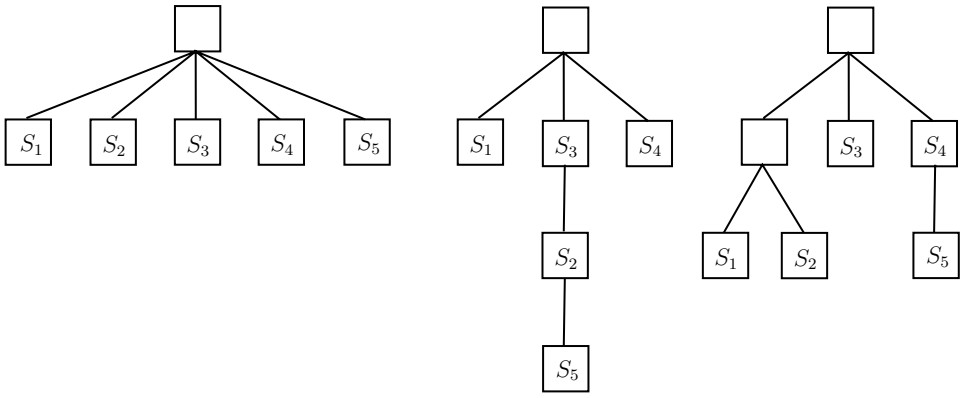
8.1.1 Arhitektura programske opreme



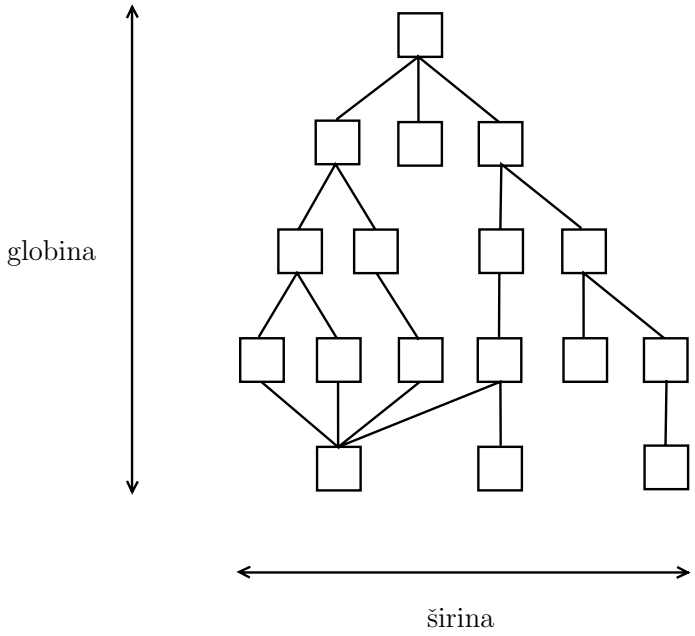
Slika 8.1: Posamezne dele problemskega področja rešimo z ustreznimi programskimi moduli.

Programska arhitektura je rezultat delitve problemskega področja, ko posameznim podproblemom ali zahtevam v specifikaciji pripišemo potrebne module programske opreme. Ta proces je simbolično prikazan na sliki 8.1. Module lahko povežemo med seboj v različne programske strukture (slika 8.2).

Po kakšnih načelih naj poteka delitev na podprobleme oziroma module in kakšna naj bo programska struktura, govorimo v podpoglavju 8.2. Programska arhitektura prikazuje nadrejenost oziroma podrejenost posameznih modulov, ne kaže pa samega vrstnega reda procesiranja, odločitev ali ponavljanj operacij. Med moduli so lahko različni odnosi; modul A lahko vsebuje modul B, modul A sledi modulu B, modul A posreduje podatke modulu B, ali modul A uporablja modul B. Programsko arhitekturo lahko pokažemo z grafi ali drugimi strukturnimi diagrami (npr. z Warnierovimi diagrami), ki prikazujejo odnos med moduli. Grafi naj imajo module razporejene po ravneh, tako da moduli uporabljajo le module na nižjih ravneh. Programsko arhitekturo lahko opišemo z nekaj preprostimi merami, kot sta *globina* in *širina* grafa ter *razvejitev (fan-out)* in *združitev (fan-in)* modulov (slika 8.3).



Slika 8.2: Programske module lahko med seboj povežemo v različne programske strukture.



Slika 8.3: Programsko strukturo opišemo z njeno globino, širino, razvejitvijo (na sliki je največja stopnja 3) in združitvijo (na sliki je največja stopnja 4) posameznih modulov.

Grafi, ki modelirajo sistemsko strukturo, so lahko ciklični ali aciklični. Posebej so zaželene drevesne strukture. Zato je za dano sistemsko strukturo koristno oceniti odstopanje od idealne drevesne strukture. Popoln graf G_n z n vozlišči ima $n(n-1)/2$ robov. Drevo z n vozlišči pa ima $(n-1)$ robov. Za poljuben graf G z n vozlišči in e robovi izrazimo odstopanje od drevesne strukture $m(G)$ kot razmerje med dejanskim številom dodatnih robov in maksimalnim možnim številom dodatnih robov:

$$m(G) = \frac{2(e-n+1)}{(n-1)(n-2)} \tag{8.1}$$

Če je G drevo, je $m(G) = 0$. Če je G polni graf, pa je $m(G) = 1$. Vztrajanje pri čisti drevesni strukturi ni vedno možno niti smiselno. Drevesna struktura tudi ne omogoča ponovne uporabe modulov, celo v istem programu ne. Prevelika stopnja združitve opozarja na morebitno premajhno kohezijo. Prevelik porast števila modulov med dvema nivojema pa opozarja na manjkajočo abstrakcijsko raven.

8.1.2 Načrt podatkovnih struktur

Podatkovne strukture ponazarjajo logična razmerja med posameznimi podatkovnimi elementi. Struktura podatkov vpliva na organizacijo, način dostopa in povezave med njenimi elementi ter s tem na sam načrt postopkov procesiranja. Zato je v postopku načrtovanja pomembno, da poiščemo oziroma načrtujemo ustrezne podatkovne strukture. O podatkovnih strukturah so napisane cele knjige (npr. [1]). Zato na tem mestu le to, da je poleg klasičnih podatkovnih struktur (vektorji, skladi, vrste, drevesa itd.), ki so v izbranih programskih jezikih lahko celo neposredno vgrajene, pomembna zmožnost definiranja novih podatkovnih struktur. Tako kot pri programski strukturi lahko tudi podatkovne strukture obravnavamo na različnih nivojih abstrakcije (abstraktne podatkovne strukture).

8.1.3 Načrt postopkov

Potem ko smo določili programsko strukturo in ustrezne podatkovne strukture, moramo določiti še postopke procesiranja v posameznih modulih. Postopki procesiranja morajo biti nedvoumno določeni in zato opis procesiranja v naravnem jeziku ni primeren. Potrebni so bolj formalni zapisi, kot so psevdokoda, grafični in tabelarni zapisi. Teoretične osnove za načrtovanje postopkov procesiranja so bile postavljene v 60-tih letih s “strukturiranim

programiranjem" [11]. Strukturirano programiranje je izpostavilo tri osnovne elemente procesiranja: *zaporedje*, *pogoj* in *ponavljanje*, ki omogoča opis kateregakoli algoritma.

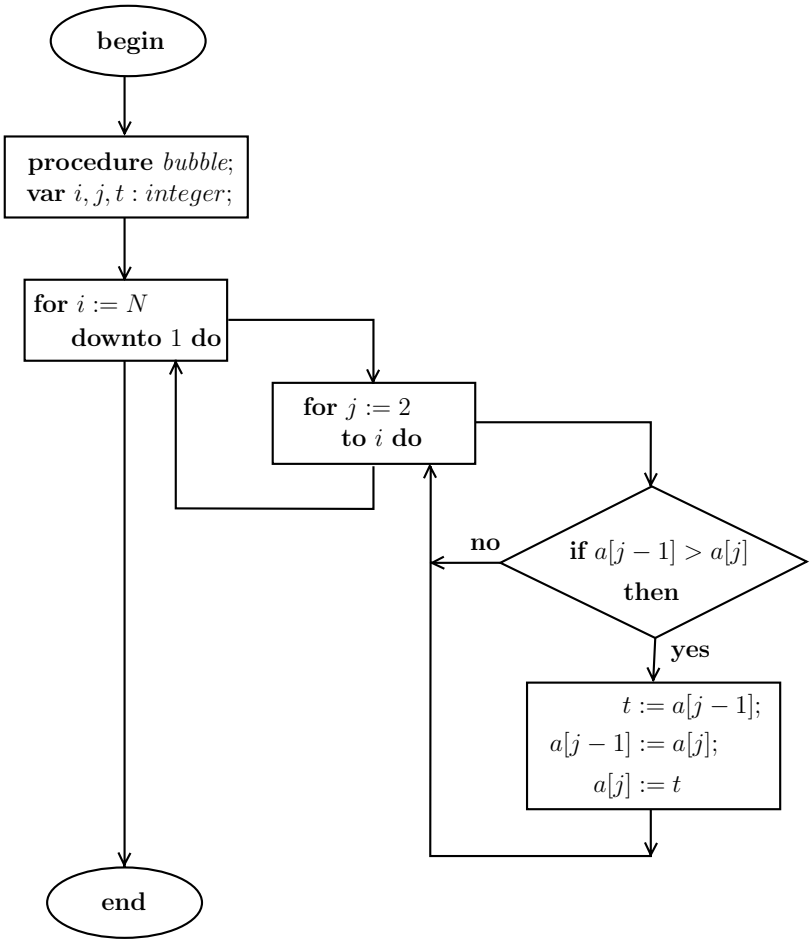
```
procedure bubble;
    var i, j, t : integer;
    begin
    for i := N downto 1 do
       for j := 2 to i do
          if a[j − 1] > a[j] then
             begin t := a[j − 1]; a[j − 1] := a[j]; a[j] := t end
    end;
```

Slika 8.4: Postopek urejanja z mehurčki (angl. bubble sort), zapisan v psevdokodi

V tako imenovani psevdokodi uporabljamo nekaj osnovnih kontrolnih struktur: *if-then-else, while-do, repeat-until* in *select-case*, ki tvorijo okostje postopka in ki jim po potrebi dodajamo opise v naravnem jeziku (slika 8.4). Zaželene so še deklaracije podatkovnih struktur in možnost definiranja in klicanja podprogramov. Kodiranje na osnovi takega načrta je enostavno, saj imajo vsi višji programski jeziki že vgrajene podobne kontrolne in podatkovne strukture. Tak načrt združuje najboljše lastnosti, ki naj bi jih imel načrt programske opreme, saj je modularen, preprost, dá se ga strojno brati in urejati.

Postopek lahko definiramo tudi z diagramom poteka (slika 8.5). Diagram poteka grafično ponazori potek procesiranja. Uporablja iste kontrolne strukture kot psevdokoda: *if-then-else, while-do, repeat-until* in *select-case*. Kontrolne strukture je možno vgnezditi eno v drugo, vendar pri obsežnih programih diagrami poteka hitro postanejo nepregledni. Hierarhičen način prikaza po nivojih omogoči večjo preglednost. Izključna uporaba strukturiranih kontrolnih struktur lahko vodi do nepotrebnih komplikacij in daljše kode. Zato je dopustno za izjemne primere uporabiti ukaz *GO TO*.

Kadar je veliko začetnih pogojev in možnih odločitev, je najpreglednejši način prikaza poteka procesiranja s pomočjo tabel (tabela 8.1).



Slika 8.5: Postopek urejanja z mehurčki (angl. bubble sort), definiran z diagramom poteka

8.2 Načrtovalska načela

8.2.1 Postopna izboljšava

Načelo postopne izboljšave kot načrtovalsko strategijo od zgoraj navzdol je predlagal Niklaus Wirth [72]. Načelo je primerno predvsem za načrtovanje majhnih programov. Arhitekturo programske opreme naj bi definirali s pomočjo postopnega razkrivanja podrobnosti. Začnemo s funkcijo, ki je izražena na visoki abstraktni ravni kot neka zahteva. O notranjem delovanju te funkcije še nič ne vemo. Na vsakem koraku izboljšave moramo funkcijo

Tabela 8.1: Potek procesiranja je lahko prikazan z odločitveno tabelo. Ko so pogoji za določeno pravilo izpolnjeni, se sproži ustrezna akcija.

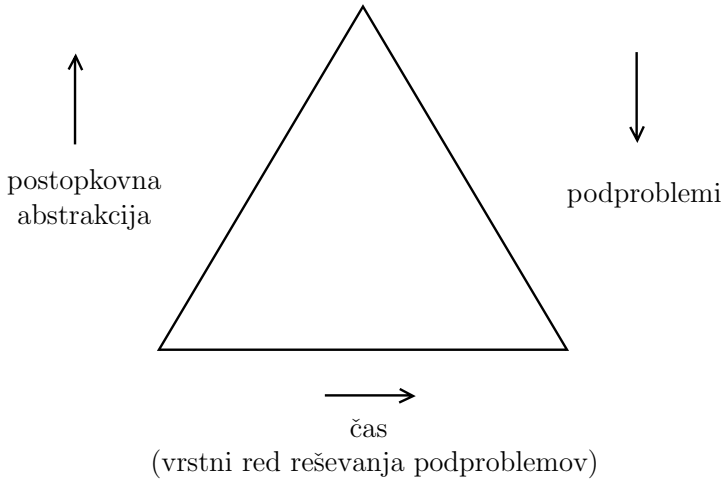
		PRAVILA								
		1	2	3	4	5	6	7	8	9
POGOJI	1. pogoj	T		F	T		F	F	F	T
	2. pogoj		F	T	F	T		T	T	F
	3. pogoj	F		F	F	T	T	F		F
	4. pogoj	F		T	F		T	T		T
	5. pogoj	T	T		F	T				T
AKCIJE	1. akcija						X			
	2. akcija	X				X			X	
	3. akcija		X		X					
	4. akcija			X				X		
	5. akcija									X

oziroma ukaz nadomestili z bolj podrobnimi ukazi. To nadaljujemo, dokler ukazi ne ustrezajo kar ukazom v izbranem programskem jeziku.

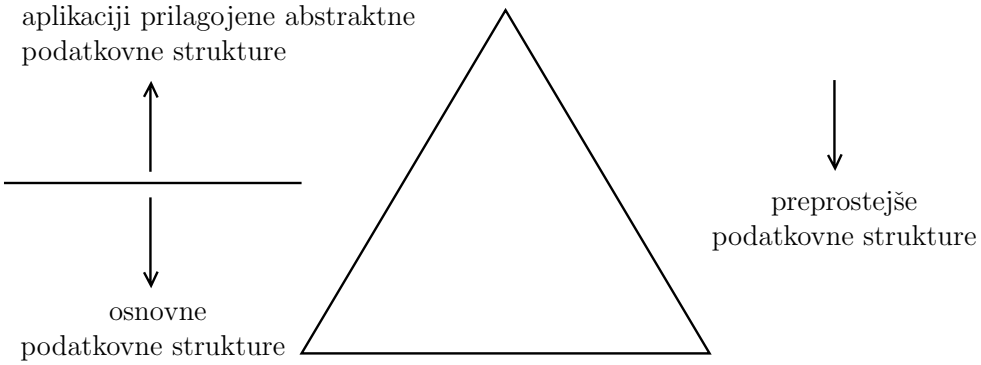
8.2.2 Abstrakcija

Abstrakcija pomeni osredotočenje na tiste lastnosti, ki so za trenutno raven obravnave bistvene, in ignoriranje ali abstrahiranje tistih podrobnosti, ki v danem trenutku niso pomembne. Zaradi kompleksnosti programske opreme jo je nujno obravnavati na različnih abstrakcijskih ravneh, saj je naenkrat nemogoče obravnavati in razumeti vse podrobnosti. Na najvišjih ravneh abstrakcije govorimo o reštivah v obliki nalog, ki jih mora programska oprema izvajati. Na nižjih ravneh abstrakcije pa za posamezne dele programske opreme že snujemo postopke za izvajanje posameznih funkcij, dokler na najnižji ravni abstrakcije ne izdelamo načrta, ki ga lahko neposredno uporabimo za kodiranje. Na primer, na višji abstraktni ravni lahko govorimo le o potrebi po urejanju nekih podatkov, šele kasneje na nižji ravni abstrakcije pa nas začne zanimati, kateri urejevalni algoritem je primerno uporabiti in kako ga implementirati. Pri razvoju programske opreme ločimo dve vrsti abstrakcije: postopkovno in podatkovno abstrakcijo.

Postopkovna abstrakcija je običajen način reševanja problemov. Da bi rešili nek zapleten problem, razmišljamo, kakšne korake moramo narediti, da bi prišli do cilja oziroma rešitve. Posamezni koraki na poti rešujejo podprobleme, katerih rešitve sestavljajo celotno rešitev. Pri tej razgraditvi



Slika 8.6: Postopkovna abstrakcija



Slika 8.7: Podatkovna abstrakcija

dobimo hierarhično strukturo, na vrhu katere je problem, ki ga rešujemo, na nižjih ravneh pa podproblemi posamezne ravni abstrakcije (slika 8.6). Na listih, najnižji ravni abstrakcije, so preprosti problemi, ki so enostavno rešljivi v izbranem programskem jeziku. Večina programskih jezikov zato nudi kontrolne strukture (pogoji, zanke ipd.), ki omogočajo postopkovno reševanje problemov. Najenostavnejša oblika rešitve problemov, ki jih razgradimo na postopkovni način abstrakcije, je vhodno-izhodno procesiranje, ko si posamezni moduli podajajo podatke. Tako zasnovane programe pa je težko spreminjati in prilagajati, saj morajo biti moduli med seboj zelo usklajeni, predvsem glede strukture in zapisa podatkov.

Podatkovna abstrakcija omogoča razgraditev problemov na način, ki pripelje do bolj neodvisnih modulov. Medtem ko postopkovna abstrakcija išče hierarhijo v kontrolni strukturi programa; katere korake je potrebno izvesti in v kakšnem vrstnem redu, pa podatkovna abstrakcija išče hierarhijo v podatkih. Programski jeziki imajo že vgrajene nekatere preproste podatkovne strukture. Iz teh je možno zgraditi bolj kompleksne, abstraktne podatkovne strukture, ki so prilagojene podatkom problema, ki ga rešujemo (slika 8.7). Tudi pri obravnavi podatkov si želimo rešiti podrobnosti, ki za določeno raven obravnave niso pomembne. Namesto, da bi se za določeno obliko predstavitve podatkov odločili že na najvišji ravni obravnave problema in to rešitev vsilili oziroma razkrili vsem modulom, podrobnosti o dejanski podatkovni strukturi raje skrijemo v poseben modul. Drugim modulom je potrebno vedeti le to, da do podatkov lahko dostopajo s pomočjo tega posebnega modula. Podatkovna abstrakcija je zato značilen primer načela skrivanja informacij.

Starejši programski jeziki (npr. Fortran, Cobol) so nudili strukture predvsem za proceduralno abstrakcijo. Novejši (npr. Ada, Modula-2, Oberon, C++) pa omogočajo tudi podatkovno abstrakcijo. Načelo podatkovne abstrakcije pri načrtovanju programske opreme je vodilo do objektno orientiranega načrtovanja, ki ga obravnavamo v podpoglavju 8.3.4.

Tretja vrsta abstrakcije je *kontrolna abstrakcija*. Kontrolna abstrakcija nam omogoča abstrahiranje kontrolnih rešitev. Taki mehanizmi so koristni predvsem za sisteme v realnem času in sisteme s vzporednim procesiranjem. Primer kontrolne abstrakcije je sinhronizacija procesiranja s pomočjo semaforjev [73].

8.2.3 Modularnost

Programska arhitektura udejanja načelo modularnosti, saj sistem razdeli na posamezne module in določi, kako so ti moduli med seboj povezani. Modularnost omogoča razumevanje kompleksnih sistemov, saj naenkrat ne moremo imeti v glavi velikega števila spremenjivk, odločitev in različnih poti skozi programsko kodo.

Za reševanje problema p s kompleksnostjo $C(p)$ potrebujemo delo $E(p)$. Če za dva problema p_1 in p_2 velja:

$$C(p_1) > C(p_2) \tag{8.2}$$

sledi:

$$E(p_1) > E(p_2). \tag{8.3}$$

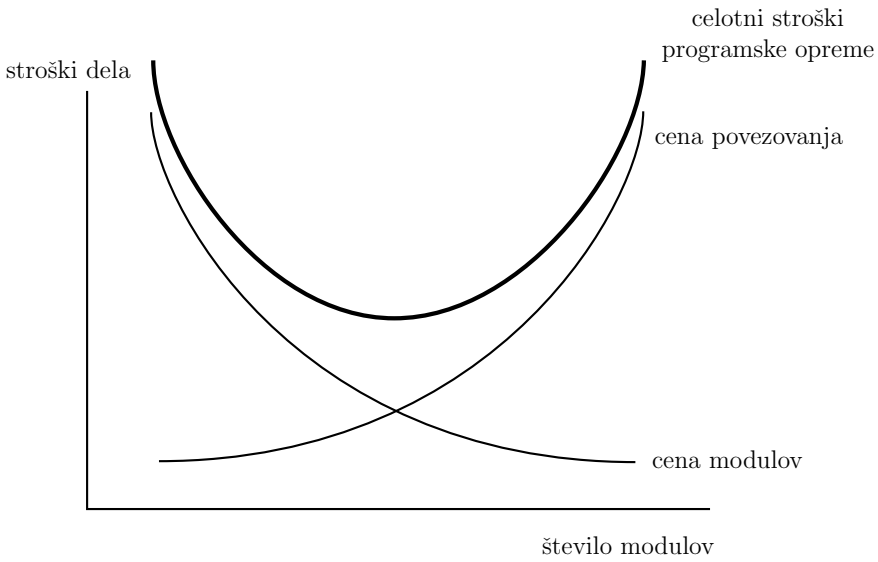
Za rešitev kompleksnejših problemov res potrebujemo več dela. Psihologi so pri preučevanju, kako ljudje rešujemo probleme, odkrili tudi:

$$C(p_1 + p_2) > C(p_1) + C(p_2). \tag{8.4}$$

Navidezna težavnost kombiniranega problema je res večja, kot če ju obravnavamo vsakega posebej. Iz enačb 8.2, 8.3 in 8.4 sledi:

$$E(p_1 + p_2) > E(p_1) + E(p_2). \tag{8.5}$$

Znano načelo "deli in vladaj" za reševanje problemov torej drži. Kompleksne probleme lažje rešimo, če jih razdelimo na obvladljive dele. To velja tudi za programsko opreme in njene module.



Slika 8.8: Cena izdelave programske opreme glede na velikost modulov

Kakšna pa je primerna velikost podproblemov oziroma modulov? Sestavljanje rešitve glavnega problema iz delnih rešitev posameznih podproblemov namreč ni zastonj. Tudi programske module moramo med seboj povezati z ustreznimi vmesniki. Pri določanju velikosti modulov si moramo zato prizadevati za najnižjo ceno (slika 8.8).

Vrste modulov

Module razlikujemo glede na:

1. **čas inkorporacije** — modul se lahko vključi že med prevajanjem ali kasneje med povezovanjem programa,
2. **aktivacijski mehanizem** — običajno je to klicanje modula, v sistemih z realnim časom pa lahko modul aktivira tudi zunanji dogodek (angl. interrupt),
3. **obliko kontrole**, kjer ločimo:
   (a) **sekvenčne** module — izvajanje programa se prenese na klicani modul, ki se v celoti izvede,
   (b) **inkrementalne** module — izvajanje modula se lahko prekine in kasneje nadaljuje na točki prekinitve,
   (c) **vzporedne** module, ki se v večprocesorskih sistemih izvajajo vzporedno, na enoprocesorskih pa se vzporedno izvajanje simulira.

Notranja enotnost modulov

Notranja enotnost (angl. cohesion) modula deluje kot lepilo, ki elemente modula drži skupaj. Bolj kot je modul notranje trden, boljši je. Po kohezijski moči, od najšibkejše do najmočnejše, ločimo naslednje stopnje kohezije [74]:

1. **naključna** — elementi modula so zbrani povsem naključno,
2. **logična** — v modulu so zbrane funkcije, ki so med seboj logično povezane, na primer vse funkcije uporabniškega vmesnika,
3. **časovna** — v modulu so zbrane vse funkcije, ki se morajo izvesti ob istem trenutku (npr. inicializacije),
4. **komunikacijska** — v modulu je več funkcij, ki uporabljajo iste zunanje podatke,
5. **zaporedna** — elementi v modulu se morajo izvesti v določenem vrstnem redu,
6. **funkcijska** — vsi elementi modula služijo eni sami funkciji.

Notranjo enotnost načrtovanega modula lahko enostavno preizkusimo tako, da namen modula opišemo z enim stavkom [62]. Če ima stavek več kot en glagol ali nekaj našteva, modul opravlja več kot eno samo funkcijo. Če so v stavku besede, ki označujejo vrstni red ali časovno zaporedje, ima modul časovno ali zaporedno notranjo enotnost.

Pri načrtovanju ni potrebno natančno določiti stopnje kohezije, temveč stremeti za notranje čim enotnejšimi moduli.

Soodvisnost modulov

Soodvisnost (angl. coupling) je mera povezanosti med moduli. Večja ko je soodvisnost med moduli, slabša je programska struktura. Po soodvisnosti (angl. coupling), od najšibkejše do najmočnejše, ločimo naslednje stopnje:

1. **podatkovna** — med moduli se prenašajo le enostavni podatki,
2. **referenčna** (angl. stamp) — med moduli se prenašajo cele podatkovne strukture, ki mora zato biti skupna tem modulom,
3. **kontrolna** — modul direktno kontrolira izvajanje drugega modula, tako da mu posreduje kontrolno informacijo (angl. flag),
4. **zunanja** — modul (npr. vhodno-izhodni) je povezan z določeno fizično napravo ali vezan na določen komunikacijski protokol,
5. **skupna** (angl. common) — moduli uporabljajo iste globalne podatkovne strukture (COMMON v Fortranu ali external v C-ju),
6. **vsebinska** — modul direktno vpliva na delovanje ali spreminja podatke drugega modula.

Večja ali manjša soodvisnost modulov je rezultat načrtovanja sistemske arhitekture. Določenim vrstam soodvisnosti ni možno ubežati, kot je na primer zunanja soodvisnost, primerno pa jih je čim bolj omejiti.

Notranja enotnost modulov in njihova soodvisnost sta nasprotni lastnosti. Če je modul notranje trden, je tudi njegova soodvisnost majhna in obratno. Veliko notranjo enotnost in majhno soodvisnost dosežemo predvsem tako, da modulom načrtujemo čim bolj preproste vmesnike. Med moduli naj bi sploh prišlo do izmenjave informacij le kot posledica ustreznega klica. Tako načrtovani moduli imajo številne prednosti:

- sodelovanje med programerji je lažje,
- manj verjetno je, da bi spremembe v enem modulu vplivale na druge,
- lažje je ponovno uporabiti module v drugih sistemih,
- delovanje modulov je lažje razumeti,
- izkušnje kažejo, da je v takih modulih manj napak.

Čeprav je programski sistem načrtovan modularno, se včasih lahko zaradi hitrosti ali omejenega spomina zakodira na “monolitni” način. Na tak način je zakodirana predvsem programska oprema za mikroprocesorje, vgrajene v druge izdelke, ali sistemi, ki delujejo v realnem času.

8.2.4 Skrivanje informacij

Načelo skrivanja informacij [48] nam pomaga pri delitvi programske rešitve na module. Moduli naj bi bili tako načrtovani, da so informacije o procesiranju in podatkovnih strukturah, ki nastopajo v modulu, *nedostopne* vsem modulom, ki teh informacij ne potrebujejo. Princip skrivanja informacij je zato tesno povezan z načeli abstrakcije ter kohezije in soodvisnosti modulov. Če modul skriva neko informacijo o svojem delovanju, lahko uporabnik tega modula to informacijo odmisli ali abstrahira. Ker je informacija skrita, je tudi ne morejo uporabiti drugi moduli, kar zmanjša nevarnost njihove soodvisnosti. Če med razvojem uporabljamo načelo skrivanja informacij, bo tak sistem lažje testirati in kasneje tudi vzdrževati.

8.3 Načrtovalske metode

Metoda načrtovanja programske opreme sestoji iz množice navodil, načrtovalskih izkušenj in postopkov. Za vsako metodo je značilna tudi notacija, s katero izrazimo rezultat načrtovanja. Vse skupaj omogoča organiziranje in strukturiranje načrtovanja.

Obstaja cela vrsta načrtovalskih metod (preko 30 [65]). Nekatere metode predpisujejo predvsem notacijo, druge pa tudi natančno določajo sam postopek načrtovanja. Nekatere metode so osredotočene le na načrtovanje, druge so zopet del širše metodologije, ki zajema celoten razvoj programske opreme. Številne metode podpirajo tudi posebna orodja CASE.

Podobno kot pri analizi zahtev lahko metode načrtovanja razdelimo po osnovnem pristopu. Smiselno je, da je metoda načrtovanja usklajena z metodo analize, na primer na osnovi pretoka podatkov ali na osnovi strukture podatkov.

8.3.1 Funkcijska dekompozicija

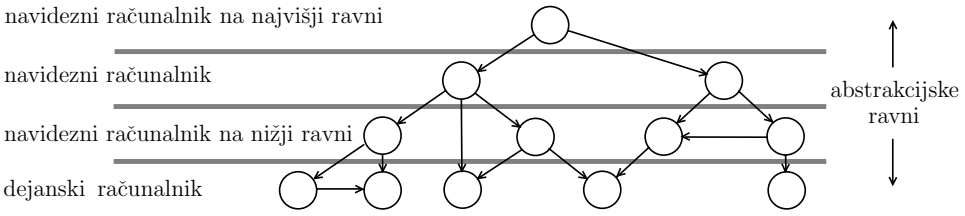
Funkcijska dekompozija je bolj filozofija načrtovanja kot metoda, ki bi natančno predpisovala postopek načrtovanja. Funkcijo, ki jo želimo implementirati, razdelimo na podfunkcije, ki vsaka rešuje del problema. Vsako podfunkcijo lahko zopet delimo naprej na enak način. Pri tem uporabljamo načelo “deli in vladaj”. Tako delitev nadaljujemo, dokler ne pridemo do osnovnih funkcij in struktur izbranega programskega jezika. Med načrtovanjem moramo zato povezati zahteve oziroma funkcije, določene med analizo z osnovno množico gradnikov programske opreme. Za povezovanje teh dveh

koncev obstajata dva osnovna načina: načrtovanje od zgoraj navzdol in od spodaj navzgor.

Načrtovanje od zgoraj navzdol začne z delitvijo glavnih funkcij programske opreme in to delitev ponavlja. Tak način je možen le, če so vse glavne funkcije sistema natančno določene.

Načrtovanje od spodaj navzgor iz osnovnih gradnikov s pomočjo abstrakcije postopoma gradi glavne funkcije. Ta način je sicer bolj prilagodljiv, vendar lahko svoj cilj (sistemske funkcije) zgreši.

Oba našteta načina načrtovanja v čisti obliki le redko uporabljamo, saj načrtovanje ni povsem racionalen proces. Načrtovalci delamo napake, uporabniki ne vedo čisto natančno, kaj želijo, prihaja do sprememb, saj šele med načrtovanjem odkrivamo dodatne informacije. Načrtovanje je zato podobno igrači "yo-yo" — načrtovalci se po potrebi in svojem nagibu sprehajamo med različnimi abstrakcijskimi ravnmi. Idejo, ki se nam porodi, izpeljemo in jo preizkusimo, jo zavrnemo in poskusimo z novo. Funkcije lahko delimo po postopkovnem ali podatkovnem abstrakcijskem kriteriju.



Slika 8.9: Navidezni računalniki predstavljajo posamezne abstrakcijske ravni v sistemski strukturi.

Pri načrtovanju je zato potrebno upoštevati vsa dobra načrtovalska načela. Parnas [49] predlaga, da na začetku identificiramo minimalno podmnožico funkcij, ki jo po potrebi postopno širimo. Zgradili naj bi hierarhično sistemsko arhitekturo, ki jo sestavljajo posamezne abstrakcijske ravni (slika 8.9). Vsaka abstracijska raven deluje kot navidezen računalnik (angl. virtual machine). Osnovne operacije navideznega računalnika so implementirane na nižjih abstrakcijskih ravneh. Vsak navidezni računalnik lahko uporablja le elemente na isti ravni ali na nižjih ravneh. Navidezni računalniki, ki so na višji stopnji abstrakcije, so bližje dejanskim uporabniškim funkcijam. Navidezni računalniki na nižjih stopnjah pa so blizu dejanskemu računalniku, na katerem se bo načrtovana programska oprema dejansko izvajala.

8.3.2 Načrtovanje na osnovi pretoka podatkov

Načrtovanje nadaljujemo tam, kjer se je končala analiza na osnovi pretoka podatkov [74]. Iz diagrama pretoka podatkov moramo izpeljati načrt programske arhitekture. Diagram pretoka podatkov ima lahko enega od dveh možnih značajev:

transformacijski značaj — podatki se med pretokom skozi sistem le transformirajo,

transakcijski značaj — v določeni tokčki sistema (transakcijski center) se poti, po katerih podatki tečejo, ločijo.

Glede na značaj diagrama pretoka podatkov obstajata dva postopka izpeljave načrta programske arhitekture.

Če ima diagram pretoka podatkov transformacijski značaj, načrtovanje poteka po naslednjih korakih (slika 8.10):

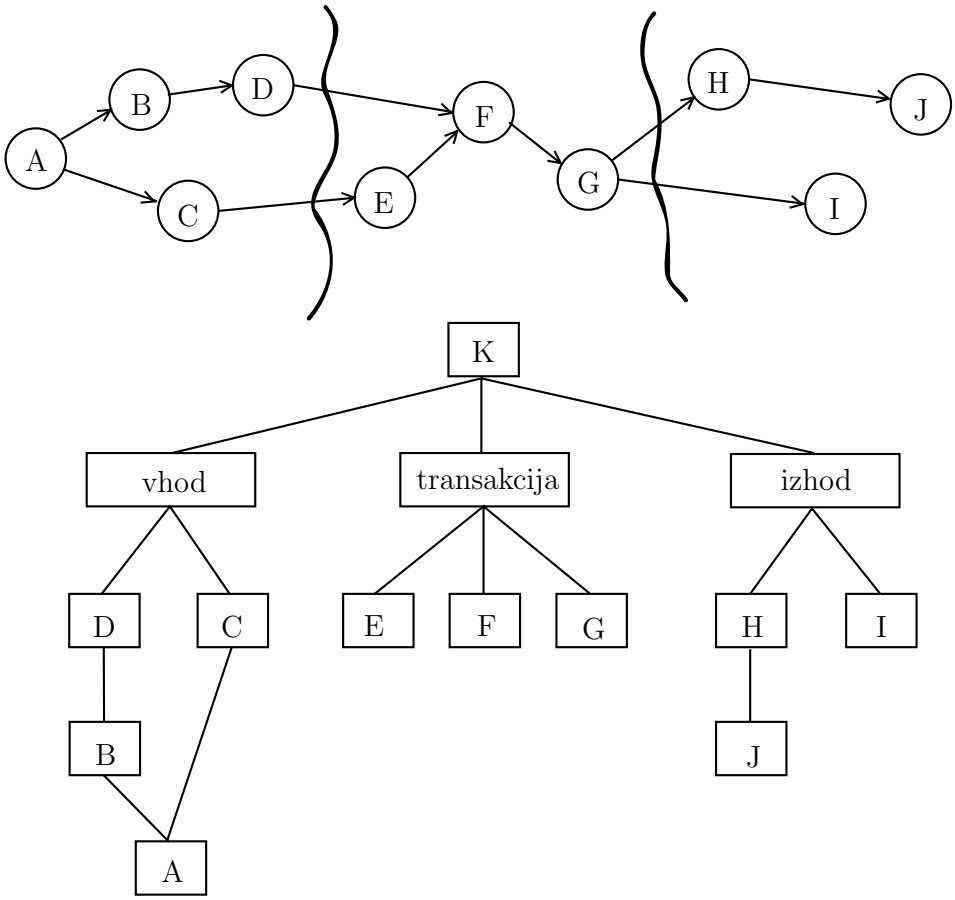
- Razmejiti je potrebno vhodne in izhodne transformacije od osrednjih transformacij. Vhodne transformacije so tiste, ki vhodne informacije transformirajo v neko notranjo obliko. V jedru programske opreme nato osrednje transformacije poskrbe za glavno transformacijo podatkov. Izhodne transformacije pa nato zopet podatke spremene v izhodni format.

- Diagram pretoka podatkov preslikamo v programsko strukturo. Na vrhu je glavni kontrolni modul, ki so mu podrejeni še trije kontrolni moduli; za koordiniranje sprejemanja vhodnih podatkov, za koordiniranje transformacijskega centra in za koordiniranje produkcije izhodnih podatkov.

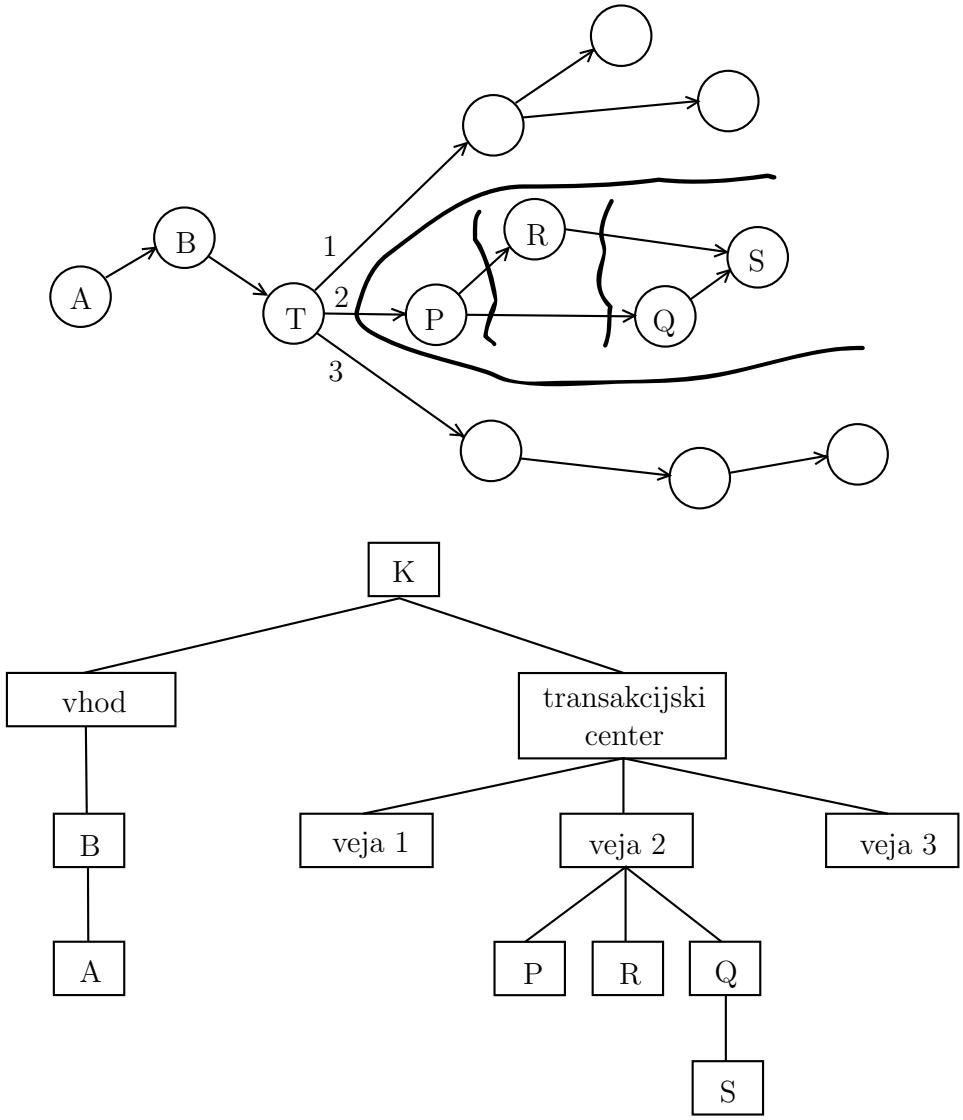
- Preslikavo diagrama poteka podatkov v programsko strukturo nadaljujemo tako, da vsaki transakciji pripišemo ustrezen modul v programski arhitekturi. Module pa med seboj povežemo na naslednji način. Postavimo se v transakcijsko središče diagrama poteka podatkov. Premikamo se proti vhodu oziroma izhodu in transakcije oziroma ustrezne module vnašamo po tem vrstnem redu v programsko arhitekturo.

- Po potrebi tako izpeljano programsko arhitekturo še izboljšamo z uporabo načrtovalskih načel. Module lahko ločujemo ali združujemo, da bi dosegli večjo notranjo enotnost in znižali soodvisnost.

Če ima diagram pretoka transakcijsko središče, pa njegova preslikava v programsko arhitekturo poteka na naslednji način (slika 8.11):



Slika 8.10: Sistemska struktura, izpeljana iz diagrama pretoka podatkov, ki ima transformacijski značaj. Na diagramu pretoka podatkov so med seboj ločene vhodne, osrednje in izhodne transformacije.



Slika 8.11: Sistemska struktura, izpeljana iz diagrama pretoka podatkov, ki ima transakcijski značaj. Tri ločene podatkovne poti vodijo iz transakcijskega centra. Na srednji poti je označena delitev na vhodne, osrednje in izhodne transformacije.

- Poiščemo transakcijski center (T) in ločimo med seboj različne poti procesiranja, ki izvirajo iz transakcijskega centra.
- Diagram pretoka podatkov preslikamo v programsko strukturo. Na vrhu je glavni kontrolni modul, ki sta mu podrejena dva kontrolna modula; za koordiniranje sprejemanja vhodnih podatkov in za koordiniranje transakcijskega centra.
- Preslikavo diagrama poteka podatkov v programsko strukturo nadaljujemo tako, da vsaki transakciji pripišemo ustrezen modul v programski arhitekturi. Povezavo modulov, ki ustrezajo transformacijam na vhodnem podatkovnem toku, določimo na enak način kot prej. Transakcijskemu kontrolnemu modulu podredimo še poseben kontrolni modul za vsako pot, ki vodi iz transakcijskega centra. Vsako od teh poti obravnavamo nato kot poseben diagram poteka s transformacijskim značajem. Module, ki ustrezajo posameznim transformacijam, zato tudi razporedimo tako kot pri načrtovanju z diagramom, ki ima transformacijski značaj.
- Po potrebi tudi to programsko arhitekturo še izboljšamo s pomočjo načrtovalskih načel.

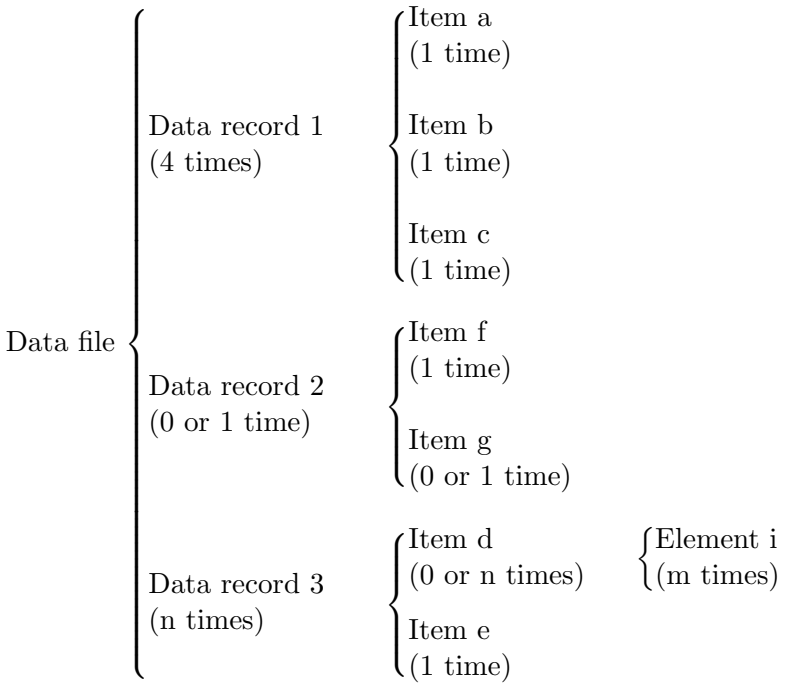
8.3.3 Načrtovanje na osnovi strukture podatkov

Pri metodah načrtovanja na osnovi strukture podatkov je načrtovanje tudi tesno povezano z analizo. Izmed številnih metod bomo na kratko predstavili metodo LCP (Logical Construction of Programs) [69], ki temelji na uporabi Warnierovih diagramov, in metodo JSD (Jackson System Development) [28], ki jo je Jackson razvil iz metode JSP (Jackson Structured Programming).

Metoda LCP

Metoda LCP zahteva, da vhodne in izhodne podatke predstavimo z Warnierovimi diagrami, kot na sliki 8.12. Ker metoda predpostavlja, da so programi, tako kot vhodni in izhodni podatki, le datoteke, metoda LCP v naslednjem koraku zahteva, da tudi procesiranje predstavimo z Warnierovimi diagrami. Hierarhijo procesiranja izpeljemo iz strukture vhodnih podatkov. Vhodno datoteko s slike 8.12 lahko procesira program, ki je prikazan na sliki 8.13.

Warnierov diagram procesiranja lahko enostavno spremenimo v običajnejši prikaz procesiranja, na primer v diagram poteka. Ponavljanje v Warnierovem diagramu spremenimo v kontrolno strukturo `repeat-until`, pogoj



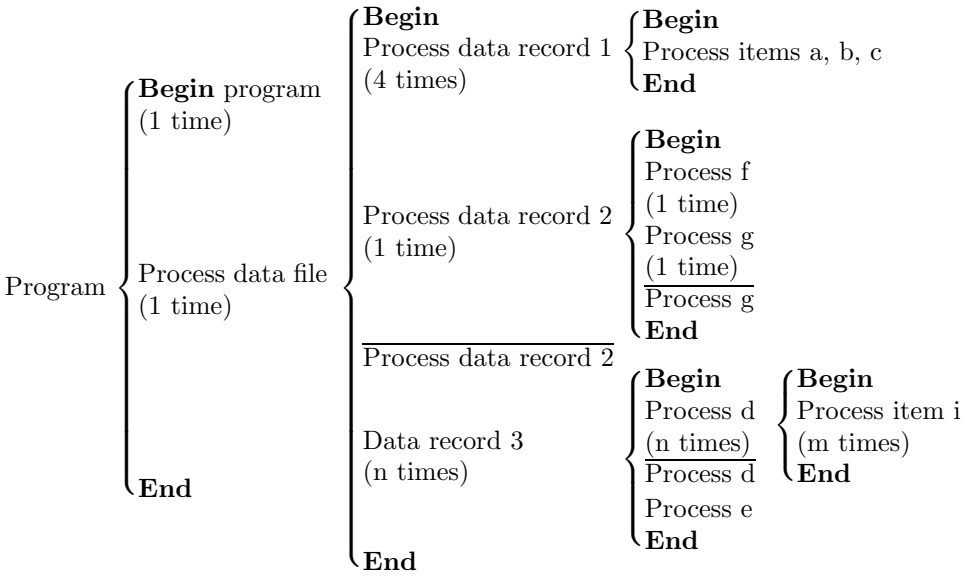
Slika 8.12: Warnierov diagram. Podatkovna datoteka vsebuje tri zapise; 1, 2 in 3, ki nastopajo 4-krat, 0 ali 1-krat ter n-krat. Zapis 1 vsebuje elemente a, b in c, ki vsi nastopajo po 1-krat. Zapis 2 vsebuje elementa f, ki nastopa 1-krat, in g, ki nastopa 0- ali 1-krat. Zapis 3 vsebuje elementa d, ki nastopa 0 ali n-krat, in e, ki nastopa 1-krat. Element d je sestavljen iz podelementov i, ki nastopajo m-krat.

pa v `if-then-else`. Warnier je poleg tega razvil metodo, poimenoval jo je "podrobna organizacija" (detailed organization), ki omogoča specifikacijo podrobnih ukazov iz diagrama poteka.

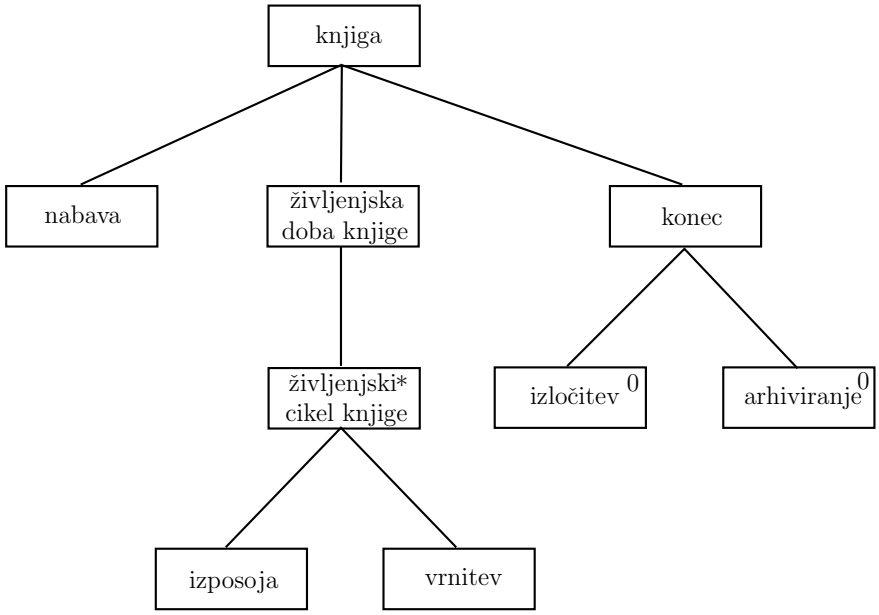
Metoda JSD

Metoda JSD (Jackson System Development) [28] predpisuje tri korake v razvoju programske opreme:

1. **Modeliranje.** Problem opišemo tako, da identificiramo osnovne enote (entitete) in akcije. JSD modelira problemsko področje kot množico enot in objektov problemskega področja, ki sodelujejo v dogodkih ali akcijah. Za vsako enoto modeliramo proces, ki prikazuje njen življenjski cikel. Akcije so dogodki, povezani s to enoto.

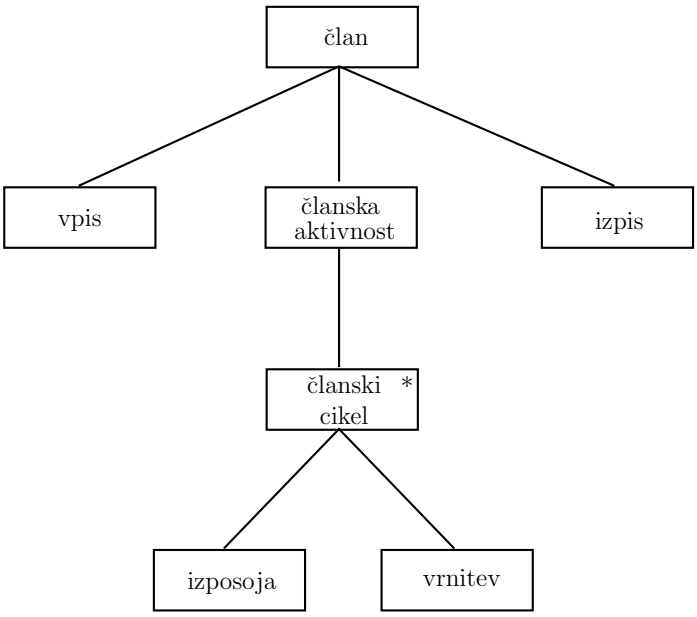


Slika 8.13: Hierarhija procesiranja za vhodne podatke s slike 8.12



Slika 8.14: Diagram strukture procesiranja za enoto `knjiga` ($*$ predstavlja ponavljanje akcije, 0 pa izbiro).

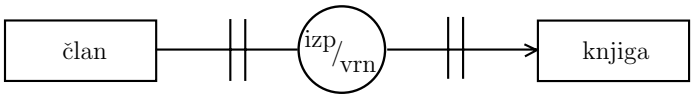
Kot primer obravnavajmo avtomatizacijo knjižnice. Dve najvažnejši enoti v tem načrtu sta knjige in člani knjižnice. Na sliki 8.14 je prikazan življenjski cikel enote "knjiga". Življenjski cikel knjige se začne z njeno nabavo. Nato se jo izposoja in vrača. Življenski cikel se konča, ko se knjigo bodisi izloči iz knjižnice (gre med star papir, se jo odproda ipd.) ali shrani v arhiv. Diagram strukture procesiranja je podoben diagramom končnih stanj (angl. finite state diagram), le da v diagramu strukture procesiranja vozlišča predstavljajo prehode med stanji, povezave pa stanja enote, ki jo diagram modelira.



Slika 8.15: Diagram strukture procesiranja za enoto `član`

Na podoben način je modelirana enota "član". Član postaneš z vpisom, člani si lahko izposojajo knjige, dokler se ne izpišejo. V diagramu strukture procesiranja so na listih drevesa akcije, ki jih ni več možno deliti naprej. Parametri vsake akcije so atributi. Za akcijo "nabava" so atributi: ISBN, datum nabave, naslov knjige, avtorji knjige itd. Enote imajo tudi atribute. Atributi enote "knjiga" so poleg atributov "nabave" še: identifikacija, status knjige (v knjižnici ali izposojena). Vsako enoto lahko obravnavamo kot poseben dolgoročen proces. V našem primeru ima vsaka knjiga in vsak član svoj življenjski cikel.

2. **Povezovanje.** Narediti moramo model, ki celoten sistem prikazuje



Slika 8.16: Pretok podatkov med enotama `član` in `knjiga`

kot omrežje sočasnih procesov in enot, ki med seboj komunicirajo s sporočili. Akciji "izposoja" in "vrnitev" nastopata v življenjskih ciklih "knjige" in "člana". Skupne akcije pomenijo, da je ustrezni enoti potrebno med seboj povezati (slika 8.16). Življenjski cikli povezanih enot se morajo med seboj sinhronizirati glede na akcije. Član si lahko izposoja in vrača knjige. To povzroči pretok sporočil ("izp/vrn" na sliki 8.16). Ker je veliko različnih "članov" in veliko "knjig", povezava med "članom" in " knjigo" na sliki predstavlja relacijo mnogih elementov z mnogimi elementi, kar je označimo z dvema črticama na povezavi.

3. **Implementacija.** Model omrežja sočasnih procesov spremenimo v sekvenčni načrt. Komunikacije med procesi (enotami) spremenimo v podprogramske klice, in jih uvrstimo na ustrezno mesto v hierarhiji, tako da posamezni procesi postanejo podrejeni drugim procesom.

8.3.4 Objektno orientirano načrtovanje

Začetki objektno orientiranih konceptov segajo v 70-ta leta, ko sta se pojavila programska jezika Simula67 in SmallTalk. Nekatere lastnosti objektno orientiranega pristopa sta uporabljali že načrtovalski metodi DSSD in JSD, ki sta omenjeni na začetku poglavja. V 80-tih letih je razvoj novih programskih jezikov, kot sta Ada in Modula, prispeval k nadaljnjemu razvoju objektno orientiranih konceptov [8]. V zadnjih nekaj letih opažamo pravi razmah raznovrstnih objektno orientiranih metod, jezikov in orodij.

Objektno orientiran pristop k analizi in načrtovanju zagovarja modularizacijo procesiranja **in** informacij. Medtem ko tradicionalne načrtovalske metode temeljijo na postopkovni abstrakciji, objektno orientirano načrtovanje temelji na podatkovni abstrakciji, skrivanju informacij in modularnosti. Namesto da bi obravnavano problemsko domeno skušali opisati s pomočjo preddefiniranih podatkovnih in kontrolnih struktur, nam objektno orientiran pristop omogoča kreiranje abstraktnih podatkovnih struktur in funkcijskih abstrakcij, ki so pisane na kožo obravnavanemu problemu. Ker so implementacijske podrobnosti določene povsem lokalno, je celoten načrt od njih

povsem neodvisen.

Osnovni pojmi objektno orientiranega pristopa so: *objekti*, njihove *lastnosti* ali *atributi*, *stanje objekta*, *obnašanje objekta*, ki je definirano z *operacijami*, ki jih objekt lahko izvaja. Objekti so med seboj hierarhično povezani v *klasifikacijsko strukturo*.

Objekti so konkretni ali abstraktni objekti realnega sveta: drevesa, psi, zakoni, pesmi ... Pes na primer ima naslednje atribute: trup, glavo, štiri noge in rep. V terminologiji programskega jezika pravimo, da objekti, ki imajo enake atribute, sodijo v isti razred. Individualni primerki razreda so instanciacije ali udejanjanje tega razreda. Na primer nemški ovčar in pekinezer sta instanciacija razreda pes, saj imata iste atribute, le njihove vrednosti so različne.

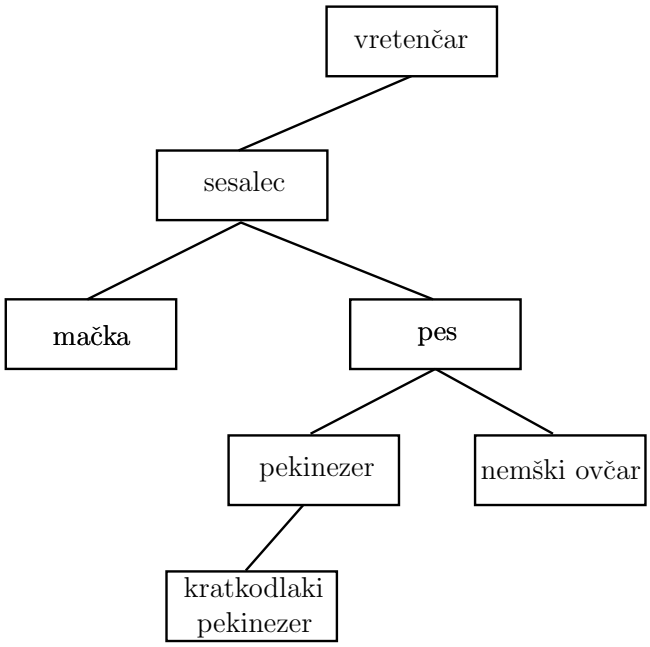
Množica atributov nekega objekta določa njegovo stanje. Tako kot v realnem svetu tudi v toku procesiranja objekti spreminjajo svoje stanje. Vsak objekt enkrat nastane, živi in na koncu ugasne. Ko objekt spreminja svoje stanje, se spreminjajo vrednosti njegovih atributov. Pes je lahko prijazen (maha z repom) ali grozi (kaže zobe). Za objekt pa ni značilno le njegovo stanje, temveč tudi operacije, ki jih objekt lahko izvaja sebi ali na drugih objektih. Obnašanje objekta je torej odvisno od teh operacij. Objekti med seboj komunicirajo s pošiljanjem *obvestil*.

Objekte lahko uvrstimo v hierarhično strukturo, ki definira njihove odnose. Objekta "pes" in "mačka" sta primerka splošnejšega objekta "sesalec". Z drugimi besedami povedano, sta pes in mačka specializaciji objekta "sesalec" oziroma sesalec je generalizacija objektov "pes" in "maček". V splošnem lahko te relacije predstavimo z acikličnim grafom, skoraj vedno pa zadošča že drevesna struktura (slika 8.17). Tiste atribute, ki so skupni različnim objektom, je zaradi ekonomičnosti smiselno definirati na najvišjem skupnem nivoju objektne hierarhije. Atribut "velikost" zato definiramo na nivoju objekta "vretenčar" namesto posebej za objekta "pes" in "maček". Prenašanje atributov s splošnejšega nivoja na nižje nivoje imenujemo *dedovanje*.

Objektno orientirana analiza in načrtovanje sestoji zato iz treh glavnih korakov:

1. **Identifikacija objektov.** Pomagamo si z opisi problemskega področja v naravnem jeziku. Našteti poskušamo glavne fizične objekte, organizacijske enote in vloge, ki jih igrajo uporabniki načrtovanega sistema.

2. **Določanje njihovih atributov in operacij.** Atributi določajo stanje objekta. Iščemo predvsem preproste in ne sestavljene atribute. Glavne operacije se nanašajo na nastanek, delovanje in prenehanje objektov,



Slika 8.17: Objektna hierarhija določa odnose med objekti.

to je na njihov življenjski cikel. Pomagamo si lahko z modelom končnih avtomatov, ki predstavlja stanja objekta. Druga vrsta operacij pa nudi informacije o stanju objekta.

3. Določanje hierarhije objektov. Hierarhija je velikokrat že določena v realnem svetu. Pogosto pa šele iskanje podobnosti med objekti pokaže na možnost vzpostavitve skupnega, bolj splošnega predhodnika. Kdaj je kakšna generalizacija (npr. pes → sesalec → vretenčar) ali specializacija (npr. pes → pekinezer → kratkodlaki pekinezer) smiselna, je odvisno od sistema, ki ga razvijamo. Pogosto so zato najbolj splošni objekti abstraktni objekti, ki nimajo nobenega primera instanciacije. Vretenčar kot posebna živalska vrsta ne obstaja, njegova instanciacija ali udejanjenje je možno le kot pes, mačka itd.

Da pridemo do končnega načrta, je potrebno naštete tri korake nekajkrat ponoviti, saj so med seboj tesno prepleteni.

Van Vliet [65] našteva naslednje prednosti objektno orientiranega načrtovanja:

- Objektno orientiran pristop je bolj naraven, saj na podoben način do-

jemamo svoje okolje. Koncepti, ki nastopajo v načrtu, neposredno ustrezajo elementom problemskega področja. Zato naročnik lažje razume načrt.

- Objektno orientiran pristop se bolj kot na eno od možnih rešitev osredotoči na strukturo celotnega problema. V funkcijski paradigmi reševanja problemov, kjer vsak modul rešuje določen podproblem, je na primer včasih težko ponovno združiti te module v celovito rešitev.

- Objektno orientiran pristop olajša prehod med analizo, načrtovanjem in kodiranjem. Fazi analize in načrtovanja je še posebej težko ločiti med seboj. Objektno hierarhijo, ki je rezultat analize, lahko direktno preslikamo v hierarhijo razredov v naši implementaciji, če uporabljamo objektno orientiran programski jezik.

- Objektno orientiran pristop vodi do bolj fleksibilnih sistemov, ki jih je lažje vzdrževati. Ker imajo realni objekti svojega dvojnika v implementaciji, lažje ugotovimo, katere module moramo spremeniti, če pride do sprememb na problemskem področju.

- Objektno orientiran pristop spodbuja ponovno uporabo programske kode. Objektno orientirana analiza identificira dejanske objekte problemskega področja, medtem ko se bolj tradicionalne načrtovalske metode osredotočijo na funkcije. Ker so v spreminjajočem se svetu objekti bolj stabilni kot funkcije, je objektno orientiran sistem manj občutljiv na spremembe okolja.

- Mehanizem dedovanja tudi prispeva k ponovni uporabnosti kode, saj se nove objekte lahko definira kot posebne primere obstoječih objektov.

- Objektno orientiran načrt naj bi utelešal vse zaželene principe dobrega načrtovanja (skrivanje informacij, abstrakcija, visoka notranja enotnost modulov, majhna soodvisnost itd.).

Van Vliet [65] hkrati tudi opozarja, da objektno orientiran pristop sam po sebi še ne zagotavlja dobrega sistema:

- Dogmatična uporaba principa enkapsulacije objektov lahko vodi do neoptimalnih rešitev, še posebej glede uporabniškega vmesnika. Dosledno objektno orientirano načrtovanje bi za vsak objekt definiralo svoj način prikazovanja tega objekta na ravni uporabniškega vmesnika. Ločeno obravnavanje uporabniškega vmesnika od ostalega dela sistema

pa se je v praksi izkazalo kot smiselno, saj so taki sistemi fleksibilnejši in uporabniški vmesnik bolj konsistenten.

- Generiranje velike množice objektov, katerih člani lahko prosto komunicirajo med seboj, lahko vodi do pretirano kompleksnih kontrolnih struktur.

8.4 Izbiranje načrtovalske metode

Vse načrtovalske metode imajo določene prednosti in slabosti. Med seboj je načrtovalske metode težko primerjati. Nobena od njih nam ne daje popolnega recepta, kako iz spiska zahtev zgraditi kvaliteten programski produkt. Izkušnje razvijalcev imajo odločilen vpliv pri vsakem razvoju programske opreme. Za vsak problem, ki ga rešujemo, poskušamo v našem spominu najti izkušnje, ki so relevantne za rešitev obravnavanega problema. Več izkušenj imamo, lažje bomo reševali različne probleme.

Nekatere metode načrtovanja programske opreme bolj natančno določajo postopek reševanja. Medtem ko je metoda funkcionalne dekompozicije zelo splošna in zahteva veliko izkušenj, metoda JSP na drugi strani dokaj natančno določa, kako naj poteka načrtovanje. Metoda JSP je uspešna predvsem takrat, ko je struktura podatkov že vnaprej določena. Kadar je potrebno definirati strukturo podatkov, sta primerni metoda JSD in objektno orientirane metode. Metode, ki temeljijo na pretoku podatkov, so primerne predvsem za avtomatizacijo obstoječih ročnih sistemov, ko uradniki obdelujejo vhodne informacije in jih predajajo drugim uradnikom v nadaljnjo obdelavo. Ker je potrebno tako rekoč vso programsko opremo kasneje vzdrževati, kar pomeni, da mora biti programska oprema fleksibilna, razumljiva in modularna, je tudi to pomemben kriterij izbire načrtovalske metode. Na izbiro pa vplivajo tudi naslednji faktorji:

- izkušnje na uporabniškem področju,
- izkušnje z načrtovalsko metodo,
- razpoložljiva orodja,
- celostna filozofija pristopa k razvoju programske opreme.

8.5 Načrt programske opreme

Osnutek načrta programske opreme naj služi kot vzorec za izdelavo načrta programske opreme.

1. Obseg programske opreme:
   (a) namen sistema,
   (b) vmesniki (programski, strojni, uporabniški),
   (c) glavne funkcije programske opreme,
   (d) zunanje podatkovne baze,
   (e) glavne omejitve pri načrtovanju.
2. Referenčna dokumentacija:
   (a) obstoječa dokumentacija programske opreme,
   (b) sistemska dokumentacija,
   (c) dokumentacija o razvojni programski opremi,
   (d) tehnične reference.
3. Opis načrta:
   (a) opis podatkov:
      i. podatkovni tok,
      ii. podatkovna struktura.
   (b) izvedena programska struktura,
   (c) vmesniki v strukturi.
4. Programski moduli (za vsak modul):
   (a) opis procesiranja,
   (b) opis vmesnika,
   (c) načrt (diagram poteka, psevdokoda),
   (d) podrejeni moduli,
   (e) organizacija podatkov,
   (f) komentarji.
5. Globalne podatkovne strukture:
   (a) zunanji podatkovni zapisi:
      i. logična struktura,
      ii. opis podatkovne strukture,
      iii. način dostopa.

   (b) globalni podatki,
   (c) korelacija med zapisi in podatki.

6. Korelacija med zahtevami in moduli
7. Testiranje:
   (a) zahteve testiranja,
   (b) strategija integracije,
   (c) posebne zahteve.
8. Način prenosa in hranjenja podatkovne opreme
9. Dodatki (preliminarna navodila za uporabo)

Poglavje 9

Kodiranje

Načrt programske opreme moramo v fazi kodiranja prevesti v izbran programski jezik, kar omogoči dejansko izvajanje programske opreme na računalniku. Če je načrt skrbno in dovolj podrobno pripravljen, je kodiranje pretežno mehansko opravilo. Vsestranska kvaliteta programske opreme je odvisna od vseh faz v razvoju programske opreme, od izbire programskega jezika in načina kodiranja pa je njena kvaliteta neposredno odvisna. Pri izbiri programskega jezika za implementacijo programske opreme igra vlogo več faktorjev. Največji vpliv na izbiro naj bi imela narava sistema, ki ga gradimo. Stopnja težavnosti prevajanja načrta v kodo je namreč odvisna od lastnosti izbranega programskega jezika, saj je splošno znano, da so za reševanje določenega problema primerni jeziki, s katerimi je lažje izraziti kontrolne in podatkovne strukture in druge zahteve problemskega področja. Vendar pa v praksi programerji izbirajo programski jezik tudi glede na njihovo seznanjenost z jeziki in izkušnje, ki jih imajo z njimi. Ker je kodiranje aktivnost, ki jo izvajajo ljudje, imajo na kakovost kode vpliv psihološke lastnosti programskih jezikov, ki lahko povečajo možnost napak med programiranjem. Programski jeziki vplivajo tudi v obratno smer — načrtovanje in naše razmišljanje o rešitvah je lahko nehote omejeno z lastnostmi izbranega programskega jezika.

Tehnične in psihološke lastnosti, v veliki meri pa seznanjenost razvijalcev z določenim jezikom, so tisti faktorji, ki v praksi odločajo o izbiri programskega jezika. Namen tega poglavja ni niti poučevanje programiranja niti pregled programskih jezikov, saj je učbenikov v ta namen veliko [33]. Našteli pa bomo nekaj lastnosti, ki vplivajo na izbiro programskega jezika in tako tudi na sam proces kodiranja.

9.1 Lastnosti programskih jezikov

Lastnosti programskih jezikov, ki imajo vpliv na proces kodiranja lahko delimo na psihološke in inženirske ali tehnične.

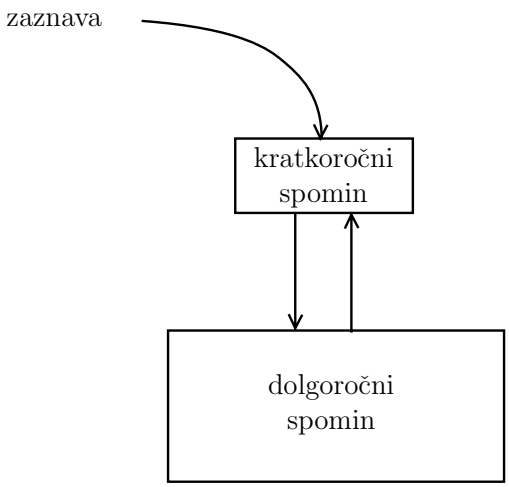
9.1.1 Psihološke lastnosti

Psihološke lastnosti programskih jezikov vplivajo na sam proces kodiranja kot človeške aktivnosti. Odločajo o naši sposobnosti učenja in uporabe programskega jezika ter vzdrževanja programske kode.

Programerji načrt programske opreme udejanijo s pomočjo programskega jezika. Zato morajo obvladati sintakso in semantiko programskega jezika. *Semanatično znanje* obsega splošne koncepte, ki niso neposredno vezani na določen programski jezik. Semantično znanje obsegajo višji koncepti, na primer kaj je urejanje, in nižji koncepti, na primer kaj je prirejanje vrednosti spremenljivki.

Sintaktično znanje obsega pravila, kako se pišejo na primer zanke, prireja vrednosti spremenljivkam, kako se posega v podatkovne strukture itd. Če nekaj časa ne programiramo v izbranem programskem jeziku, lahko njegova sintaktična pravila hitro pozabimo. Na osnovi poznavanja semantičnih struktur pa se lahko tudi hitro naučimo novega programskega jezika. Naučiti se moramo le novih sintaktičnih pravil. Učenje prvega programskega jezika je zato težje kot učenje drugega in naslednjih. To seveda ne velja, če je semantična struktura jezikov zelo različna, kot je na primer pri Pascalu in LISP-u. Sintaktično in semantično znanje nista tesno povezani. Čeprav vemo, kaj je kazalec, lahko hitro pozabimo, kako se označi v določenem programskem jeziku. Sintaktična pravila različnih jezikov zlahka pomešamo.

Za razumevanje programiranja kot človeške aktivnosti je potrebno razumeti osnovno strukturo človeškega spomina (slika 9.1). Informacije, ki jih dojemamo s pomočjo čutil, shranjujemo najprej v *kratkoročni spomin*. Kratkoročni spomin ima zelo omejeno kapaciteto. Miller [41] ocenjuje, da lahko vanj shranimo le okoli sedem enot. Z združevanjem informacij lahko manjše enote združujemo v večje enote (angl. chunking). Telefonska številka "1768 387" lahko zavzame vseh sedem mest v kratkoročnem spominu. Če pa jo spoznamo kot telefonsko številko Fakultete za računalništvo in informatiko, zavzame le eno mesto. Enote v kratkoročnem spominu služijo kot neke vrste nalepke za enoto informacij, naj bo ta enota posamezna cifra, telefonska številka ali algoritem. Ko informacije procesiramo oziroma rešujemo probleme, služi kratkoročni spomin kot *delovni spomin*. Naše znanje, izkušnje in vse, kar vemo, je shranjeno v *dolgoročnem spominu*, ki



Slika 9.1: Komponente človeškega spomina

ima skoraj neomejeno kapaciteto. Izkušeni programerji imajo svoje znanje in izkušnje organizirane v večje spominske enote, na primer proces iskanja, algoritem urejanja itd. Z njihovo pomočjo lahko zato izkušeni programerji hitreje razmišljajo o rešitvah problemov in hitreje razumejo tuj program, saj v kodi take enote kar neposredno prepoznajo, ne da bi morali pregledati vse podrobnosti [66].

Določene značilnosti programskih jezikov lahko kodiranje olajšajo ali pa povečajo možnost napak. Psihološke lastnosti so v veliki meri določene že z definicijo samega programskega jezika. Med važnejše psihološke lastnosti programskih jezikov štejemo:

Uniformnost. Stopnja konsistentne uporabe notacije je odvisna od uniformnosti programskega jezika. Na primer, če okrogle oklepaje uporabljamo v aritmetičnih izrazih, za omejitev števcev v vektorskih podatkovnih strukturah in omejitev argumentov podprogramskih klicev, taka mešana uporaba lahko poveča možnost napak pri kodiranju. Zaradi pomanjkanja simbolov v naboru znakov ASCII so avtorji programskih jezikov velikokrat prisiljeni, da iste znake uporabijo za različne operacije (ang. operator overloading). Znak "+" običajno pomeni seštevanje, lahko pa tudi unijo dveh množic, če sta argumenta množici.

Dvoumnost. Prevajalnik vsak sintaktično pravilno zapisan izraz interpretira na enak način. Če prevajalnik določa vrstni red operacij glede na vrstni red zapisa, na primer od leve proti desni, bo izraz `x =`

`a/b*c` interpretiral kot `x = (a/b) * c`. Vendar lahko isti izraz nek programer razume tudi kot `x = a/(b * c)`. Programski jeziki, ki dovoljujejo dvoumno zapisovanje aritmetičnih izrazov, s tem povzročajo večjo možnost napak.

Pogost vir dvoumnosti so tudi implicitne deklaracije podatkovnega tipa spremenljivk. V Fortranu in Basicu je prva uporaba imena tudi implicitna deklaracija tega imena. Če se ime začne s črkami I do N, ime pomeni celo število, sicer pa realno število. Če želimo odstopati od tega pravila, moramo uporabiti eksplicitno deklaracijo podatkovnega tipa.

Kompaktnost programskega jezika je odvisna od vrste in števila sintaktičnih informacij, ki si jih moramo zapomniti. Na kompaktnost programskega jezika vpliva:

- število in raznolikost kontrolnih in podatkovnih struktur, ki omogočajo jedrnato izražanje naših idej,
- vrsta besed (njihova dolžina!) in okrajšave, ki jih lahko uporabljamo,
- število vgrajenih funkcij in operacij.

Z vidika človeškega spomina je bolje, če lahko svoje ideje izražamo na čim krajši način in če jih lahko združujemo v bloke. Dolge zanke in številne razvejitve slabšajo razumljivost kode.

9.1.2 Tehnične lastnosti

Tehnične lastnosti programskih jezikov odločajo predvsem o primernosti programskega jezika za določeno vrsto problema. Osnovne lastnosti programskih jezikov, gledano s tehničnega vidika, so:

1. težavnost prevajanja načrta programske opreme v kodo,
2. učinkovitost prevajalnika,
3. prenosljivost kode,
4. ustrezna programska razvojna orodja,
5. zmožnost vzdrževanja.

9.1.3 Jezikovni konstrukti

Našteli bomo nekaj najvažnejših jezikovnih konstruktov in njihovih oblik, ki odločajo o zgoraj omenjenih psiholoških in tehničnih lastnostih.

Deklaracije. Z deklaracijami določimo, katera imena lahko uporabljamo v programu, kje se deklaracija nahaja, pa tudi območje uporabe tega imena. Nekateri programski jeziki omogočajo implicitno deklaracijo imen (ko ime prvič uporabimo), bolje pa je, kadar se zahteva eksplicitna deklaracija vseh imen. Če deklaracija določa tudi podatkovni tip imena, lahko prevajalnik kontrolira pravilno rabo imen. Tudi konverzija podatkovnih tipov je lahko implicitna (na primer, pri seštevanju celega in realnega števila se celo število spremeni v realno). Natančnejša avtomatska kontrola podatkovnih tipov je možna, kadar moramo vse spremembe eksplicitno definirati.

Podatkovni tipi in strukture. Številni programski jeziki omogočajo tudi definiranje novih podatkovnih tipov in podatkovnih struktur na osnovi že vgrajenih. Prevajalnik lahko kontrolira tudi te, uporabniško definirane podatkovne tipe in strukture. Z njihovo pomočjo lahko kodo veliko lažje prilagajamo našim zahtevam.

Kazalci. Nekateri programski jeziki omogočajo indirektno kontrolo objektov s pomočjo kazalcev. Kazalci so posebej primerni za definicijo rekurzivnih in dinamičnih struktur. Kadar na iste objekte kaže več kazalcev, lahko hitro pride do napak. Uporaba kazalcev zahteva tudi dodatno skrb pri upravljanju s spominom.

Inicializacija. Do napak lahko pride, kadar uporabimo spremenljivke, ki jim prej še nismo določili vrednosti. Nekateri programski jeziki omogočajo inicializacijo spremenljivk hkrati z njihovo deklaracijo, drugi zopet uporabljajo implicitni mehanizem inicializacije.

Kontrolne strukture. Osnovne kontrolne strukture za ponavljanje so zanke `for`, `while` in `repeat`. Medtem ko je normalno, da programer v telesu zanke `while` ali `repeat` kontrolira, kdaj se bo le-ta končala, je spreminjanje števca v telesu dotične zanke `for` recept za napake in nerazumljivost.

Osnovna kontrolna struktura za izbiranje je `if then else`. Nekateri programski jeziki imajo tudi možnost izbirati med več možnostmi s stavkom `select case`.

Veliko črnila je bilo prelitega v zvezi z uporabo oziroma prepovedjo uporabe ukaza `GOTO`. Raba ukaza `GOTO` dela kodo nepregledno in težje razumljivo, vendar pa lahko v določenih izjemnih primerih naredi kodo enostavnejšo in krajšo.

Moduli. Večina programskih jezikov omogoča definicijo programskih modulov (funkcij, podprogramov, procedur), da bi lahko sistem razdelili v logične enote. Programski jeziki se precej razlikujejo pri sintaktičnih pravilih, ki veljajo za obseg spremenljivk (lokalne/globalne, vidne/skrite). V tistih jezikih, kjer so pravila modularnosti (visoka kohezija, nizka soodvisnost) strožja, je manj možnosti za napake in je sintaktična kontrola pravilnosti lažja.

Prenos parametrov. Pri formalni definiciji procedur uporabimo formalne parametre. Ko proceduro kličemo v programu, moramo uporabiti dejanske parametre. V različnih programskih jezikih obstajajo različni mehanizmi za priredbo dejanskih vrednosti k formalnim parametrom: klic po imenu (angl. call-by-name), klic po naslovu (angl. call-by-reference), klic po vrednosti (angl. call-by-value).

Rekurzija. Nekatere algoritme se da veliko krajše in razumljivejše zapisati na rekurziven način, če to programski jezik dovoljuje. Vendar se rekurzivno zapisani algoritmi izvajajo v nekaterih primerih nekoliko dalj časa zaradi večkratnih klicev iste procedure.

Da bi izboljšali psihološke in tehnične lastnosti programskih jezikov, je sintaksa novejših programskih jezikov na splošno bolj zavezujoča. Prevajalniki in druga programska orodja lahko zato pomagajo odkriti v kodi več napak.

9.2 Specializacija programske kode

Pri kodiranju naj bi na splošno stremeli k čim bolj enostavnim in splošnim rešitvam, saj je taka koda lažje prenosljiva in omogoča lažje popravljanje ter testiranje. Koda naj bo zato le tako hitra oziroma učinkovita, kot to zahteva specifikacija in naj **ne** bo najhitrejša (najboljša) možna!

Da bi dosegli večjo hitrost izvajanja programa, moramo včasih določen del programske kode specializirati. Specializacijo začnemo z enostavnimi ukrepi, kot so poenostavitev aritmetičnih in logičnih izrazov ter optimizacija zank, kar sicer nekoliko zmanjša berljivost kode. Uporaba enostavnih podatkovnih struktur, celoštevilčnih ali Boolovih spremenljivk tudi poveča

hitrost kode. Le v nujnih primerih je smiselno splošnim rešitvam oziroma algoritmom dodajati obravnavo izjemnih primerov, ki praviloma hitro rešijo del primerov ali posebej pogoste primere. S tem se obseg kode poveča in testiranje postane zahtevnejše, saj je potrebno preveriti tudi pogoje za izjemne primere. V izjemnih primerih je potrebno dele sistema zakodirati v zbirnem jeziku ali celo uporabiti posebno strojno opremo.

9.3 Dokumentiranje programske kode

Komentarji med kodo so za hitro razumevanje kode neobhodni. Pri kodiranju je izredno pomembno sprotno dokumentiranje kode, saj je delo težje in manj natančno, če se dokumentira po opravljenem kodiranju.

Interna dokumentacija programske kode sestoji iz uvodnih komentarjev in komentarjev med kodo, važna pa je tudi logična in vizualna organizacija kode.

V *uvodni komentar* sodijo opisi programske opreme s širšega vidika, ki program oziroma modul razloži kot celoto:

1. namen ali funkcija modula,
2. za kakšno strojno opremo, operacijski sistem in konfiguracijo je namenjen,
3. ali je potrebna interakcija z uporabnikom,
4. s katerimi algoritmi so realizirane funkcije,
5. opis vmesnika (primer klica modula, opis in dovoljen obseg vrednosti argumentov)
6. spisek podrejenih modulov,
7. pomembne spremenljivke in omejitve,
8. čas izvajanja,
9. natančnost,
10. razvojna zgodovina (avtorji, spremembe, datumi).

Komentarji med kodo naj bodo le tam, kjer je kakšna posebnost. Pri vzdrževanju programske opreme je potrebno tudi ustrezno spremeniti komentarje. Napačni komentarji so še slabši kot nikakršni! Za berljivost in

razumljivost kode poskrbimo v prvi vrsti z ustrezno izbiro imen. Imena, ki sama pojasnujejo svoj pomen, imenujemo mnemonična. Kateri izraz od spodnjih treh je razumljivejši, ni potrebno posebej razlagati.

```
    d  =  v * t
 dist  =  vel * time
dist_m =  vel_m_per_sec * time_in_sec
```

S konsistentno *vizualno organizacijo* (zamikanje začetka vrstic, izpuščanje praznih vrstic, le ena izjava v vrstici itd.) je možno jasno poudariti strukturo programa. Posebni urejevalniki ali dodatki urejevalnikov za pisanje programske kode sami poskrbijo za konsistentno vizualno organizacijo. Četudi programski jezik tega ne zahteva, je dobro v vsakem modulu spremenljivke deklarirati vedno na istem mestu in po istem vrstnem redu. Uniformno organizirane module je lažje brati in spreminjati.

Celotnemu programskemu paketu naj bodo kot posebna vrsta dokumentacije priloženi *kratki testni primeri*. Z njihovo pomočjo se uporabnik lahko prepriča, če je program pravilno instaliral in preizkusi njegovo delovanje.

9.4 Uporabniški vmesniki

Pri kodiranju oziroma že pri predhodnem načrtovanju moramo veliko pozornost posvetiti vmesnikom med programsko opremo in uporabniki. Ker programsko opremo uporablja vedno več ljudi, sta enostavna uporaba ter hitro učenje postala izredno pomembna lastnost programske opreme. Od uporabniškega vmesnika je v največji meri odvisno, kako bo uporabnik sprejel programsko opremo. Na komunikacijo človek – stroj vpliva veliko faktorjev, od psiholoških, socialnih, ekonomskih, do ergonomskih in tehničnih [54].

Odnose med človekom in računalnikom lahko razjasnimo s tremi vrstami modelov [45]:

Mentalni model. Uporabnikov mentalni model je model sistema oziroma računalnika, ki si ga ustvari uporabnik na osnovi svoje izobrazbe, znanja o sistemu in problemski domeni, znanja o drugih sistemih in izkušenj. Pri interakciji s sistemom uporablja mentalni model za načrtovanje akcij, napoved in interpretacijo reakcij sistema. Mentalni model odraža uporabnikovo razumevanje sistema, kako dela in zakaj dela na določen način. Mentalni model zgradimo z učenjem, branjem dokumentacije, predvsem pa z uporabo sistema. Mentalni model

pogosto ni pravilen v tehničnem smislu. Vsebuje lahko napačne predstave in ni popoln.

Slika sistema. Uporabnikov mentalni model je osnovan na sliki sistema, ki vsebuje vse elemente sistema, s katerimi pride uporabnik v stik (fizični izgled računalnika in perifernih naprav, način in vsebina interkacije).

Konceptualni model je tehnično pravilen model računalniškega sistema, ki so ga izdelali načrtovalci in tisti, ki skrbijo za šolanje uporabnikov. Z uporabniškega vidika je konceptualni sistem popoln in konsistenten.

Pri načrtovanju uporabniških vmesnikov je osrednja naloga omogočiti čim lažjo uskladitev mentalnega in konceptualnega modela. Take sisteme se lažje in hitreje naučimo uporabljati. Če sta modela neusklajena, so napake pogoste, uporabniki pa postanejo nezadovoljni. Ker mora tipični uporabnik interaktivnega sistema opraviti določene naloge, je znanje o teh nalogah in njihovi strukturi že v človekovem spominu. Uporabniški vmesnik naj bi zato čimbolj posnemal to strukturo. Razvijalci sistemov žal strukturo uporabniškega vmesnika sestavijo po zgledu strukture tehnične implementacije. Nekateri prvi kalkulatorji Hewlett-Packard na primer namesto običajnega vrstnega reda aritmetičnih operacij (operand, operacija, operand) uporabljajo tako imenovano reverzno poljsko notacijo (operacija, prvi operand, drugi operand), ki odseva operacije v notranji podatkovni strukturi — skladu.

Glede na strukturo človeškega spomina (omejen kratkoročni spomin) naj bi se v uporabniških vmesnikih izogibali dolgim zaporedjem ukazov in menujev z velikim številom ukazov. Uporabniški vmesnik naj bo čimbolj konsistenten. Uporabimo lahko znanja in spretnosti z drugih področij. Uporabljajmo analogije (urejanje tekstov naredimo podobno pisanju s pisalnim strojem) in metafore (računalniški ekran kot delovna miza, na kateri so mape z dokumenti). S tem, da iste spretnosti lahko uporabimo v podobnih situacijah, tudi krepimo uporabnikovo samozavest.

Krivulja učenja (hitrost, s katero se naučimo nove spretnosti — angl. learning curve) je boljši pokazatelj uporabnosti sistema kot "prijaznost" uporabniškega vmesnika. Ker morajo številni uporabniki takoj začeti z uporabo sistema, hitro pridejo do določenega nivoja znanja, ki je zanje zadosten. Večina uporabnikov sistema UNIX na primer uporablja le majhno število ukazov. Za druge ukaze vedo, da obstajajo, vendar ne vedo, kako jih lahko uporabljajo (ne poznajo njihove semantike ali sintakse). Tretjo skupino pa tvorijo ukazi, za katere sploh ne vedo. Brez dodatne pomoči, šolanja ali dobre dokumentacije se ni možno naučiti popolnoma obvladati

sistem. To je značilno za večino programske opreme in njihovih uporabnikov. Nekateri sistemi namerno organizirajo svojo funkcionalnost po nivojih, tako da začetnikov ne zmede preveliko število funkcij. Dokumentacija o sistemu in sistem za pomoč morata biti strukturirana glede na strukturo nalog, ki jih s sistemom opravljamo in ne glede na strukturo samega sistema.

Kadar sistem zahteva interakcijo z ljudmi, je potrebno podatke, ki jih je vnesel uporabnik, preverjati. Kadar sistem zahteva od uporabnika nek podatek, mora sistem navesti možne izbire ali mejne vrednosti ter enote, v katerih naj bo podatek vnešen, če gre za neko veličino. Vse vnešene podatke mora sistem preverjati, če so v veljavnem obsegu ali dovoljeni obliki. Pri informacijah sestavljenih iz več enot, lahko preverjamo tudi smiselnost njihove kombinacije. Pri sistemih, ki zahtevajo vnašanje velikega števila podatkov, je pomembno delo čimbolj poenostaviti s pomočjo raznih bližnjic in možnostjo popravljanja posameznih elementov.

Velik napredek strojne in sistemske programske opreme je omogočil enostavno izdelavo grafičnih uporabniških vmesnikov, generacijo zvoka in uporabo še drugih vhodno/izhodnih naprav. Uporaba miške in koncept direktne manipulacije z objekti na računalniškem zaslonu sta danes že povsem običajna. Načrtovanje uporabniških vmesnikov je prav zaradi bogatih možnosti interkacije postalo posebno področje programskega inženirstva, o katerem obstaja veliko knjig in druge literature [57, 54].

Pri oblikovanju multimedijskih uporabniških vmesnikov se morajo razvijalci odločati o uporabi barv in zvoka ter o vizualni organizaciji objektov na zaslonu. Zato morajo oblikovalci takih interaktivnih grafičnih vmesnikov upoštevati izkušnje in dognanja psihologije. *Barve* ne smemo uporabljati le zato, da nekaj polepšamo. Barve ponudimo kot *dodaten* vir informacij, nikakor pa ne kot edini vir informacij, saj je ločljivost barv na zaslonu zelo odvisna od razsvetljave okolice, del uporabniške populacije pa je barvno slep. Barve se najlepše vidijo na sivem ozadju. Nekatere barve bolj izstopajo od drugih, toda majhne barvne površine je po barvi težko ločiti med seboj.

Za *zvok* veljajo podobna pravila. Zvok naj služi le za podkrepitev opozoril, ko se zgodi kaj nenavadnega, ali ko po daljšem času sistem zopet zahteva uporabnikovo pozornost.

Pri oblikovanju uporabniških vmesnikov je potrebno vključiti bodoče uporabnike. Ker še ne poznamo dovolj dobro delovanje človeka kot informacijsko procesnega sistema, moramo z eksperimenti preučiti reakcije uporabnikov in postopno izboljšati uporabniški vmesnik.

Poglavje 10

Testiranje

Kvaliteta programske opreme je odvisna od celotnega razvojnega procesa. Testiranje je pri razvoju osrednja aktivnost, ki preverja kvaliteto programske opreme, ni pa edina aktivnost. Danes je uveljavljeno prepričanje, da je potrebno s primernim nadzorom razvoja programske opreme napake že vnaprej *preprečevati* in pomanjkljivosti *sproti* odkrivati. Kasneje v toku razvoja programske opreme kaj spremenimo, dražja je sprememba. Med vzdrževanjem lahko dodajanje neke funkcije, ki smo jo pozabili navesti v specifikaciji programske opreme, stane celo stokrat več, kot če bi nanjo pomislili že pri specifikaciji (slika 10.1). Testiranje je zato integralni del nadzora kvalitete celotnega razvojnega cikla, pri katerem moramo poskrbeti, da je rezultat vsake razvojne faze kvaliteten in v skladu z globalnimi cilji.

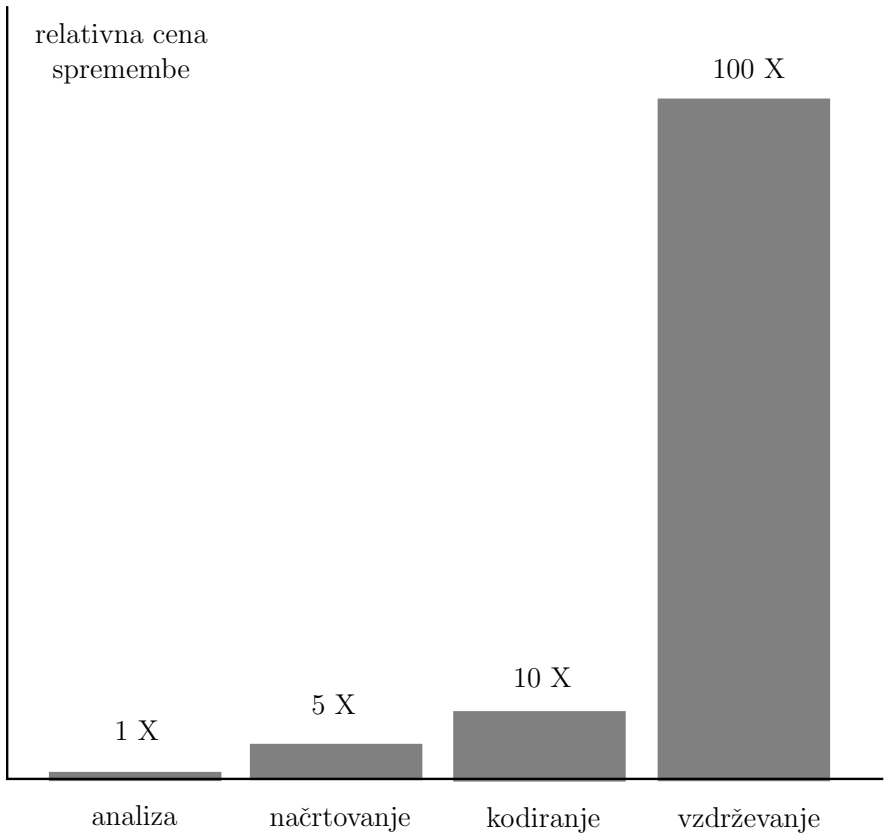
Uspešnost vsake razvojne faze ugotavljamo s posebnimi recenzijami, ki so lahko bolj ali manj formalne narave, opravijo jih lahko posamezni razvijalci ali posebne skupine za zagotavljanje kvalitete. Pomembno je, da ob vsakokratnem preverjanju ugotovimo:

1. Ali programsko opremo gradimo pravilno? Ta postopek imenujemo *verifikacija*. Verifikacija ugotavlja predvsem tehnično pravilnost zadnje faze razvoja.

2. Ali programska oprema ustreza osnovnim zahtevam? Ta postopek imenujemo *validacija*. Validacija preverja, če gradimo pravi produkt.

Testiranje je potemtakem aktivnost, ki verificira fazo kodiranja, na koncu pa še preverja, če celoten delujoč program ustreza osnovnim zahtevam, zapisanim v specifikaciji.

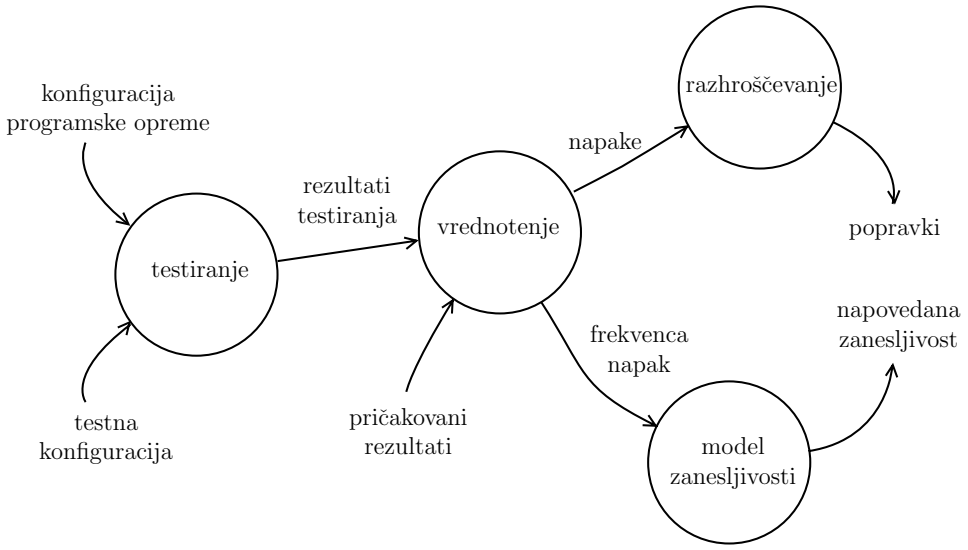
Programsko kodo testiramo tako, da jo izvajamo in preverjamo, če pri tem pride do napak. Napake moramo seveda odpraviti, analiza pogostnosti



Slika 10.1: Cena sprememb v programski opremi od faze analize do faze vzdrževanja strmo narašča.

in vrste napak pa omogoča izdelavo modela zanesljivosti (slika 10.2). Testne primere želimo izbrati tako, da bi našli čimveč napak. Dober testni primer je tisti, pri katerem je verjetnost, da se najde napaka, čim večja. *Vendar testiranje programa lahko pokaže samo na prisotnost napak, ne more pa dokazati, da napak v programu ni!* Če bi želeli zagotoviti popolno zanesljivost nekega programa, bi morali preizkusiti vse možne načine izvajanja tega programa. To pa žal v praksi ni izvedljivo, saj število možnih poti skozi program oziroma število različnih možnih načinov izvajanja programa eksponentno narašča. Iskanje napak v programski kodi je kot iskanje igle v kopici sena. Kljub temu, da testiranje ne zagotavlja popolne odsotnosti napak, s sistematičnim testiranjem večamo naše zaupanje v programsko opremo.

Če bi se zanka v programu, ki ga prikazuje diagram poteka na sliki 10.3,



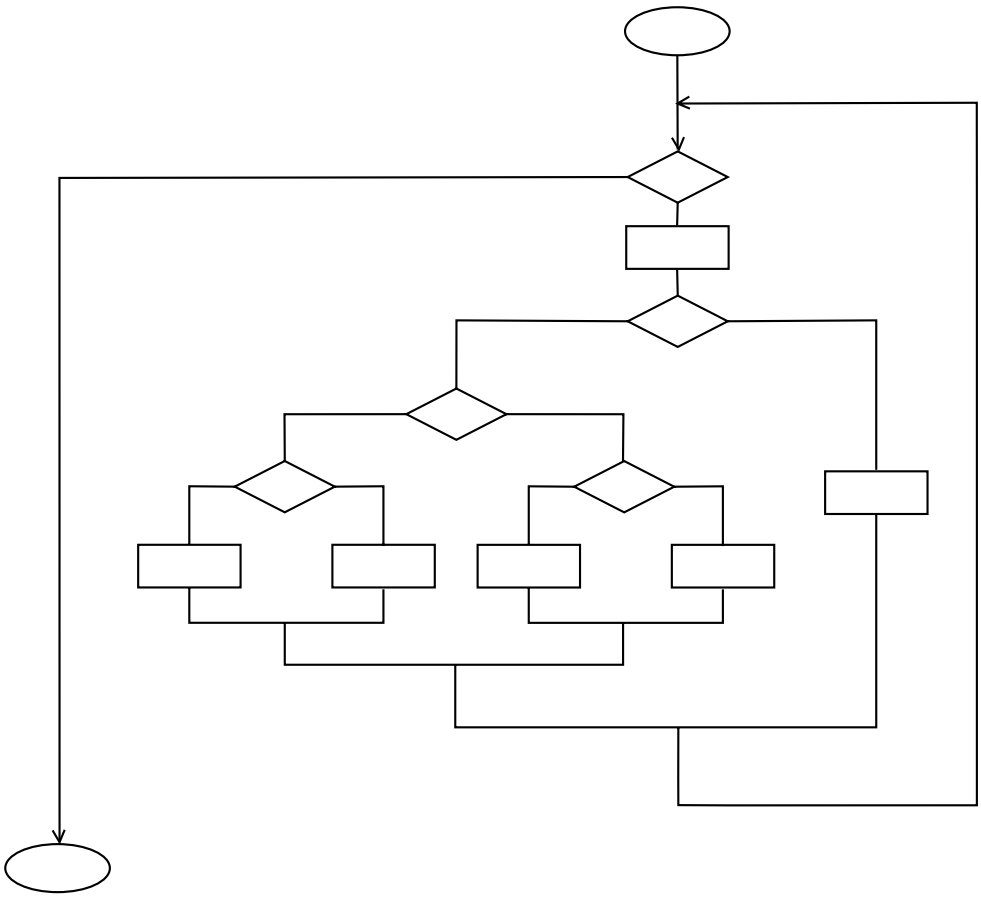
Slika 10.2: Potek testiranja programske opreme

ponovila dvajsetkrat, bi bilo število možnih načinov izvajanja programa okoli:

$$N = 5^{20}.$$

Pri vsakokratni ponovitvi zanke lahko program izbere eno od petih poti. Izbira poti pa je lahko med posameznimi ponovitvami zanke povsem neodvisna od predhodnih poti. Zato se ob vsaki ponovitvi zanke za petkrat poveča število različnih poti izvajanj celotnega programa. Če bi želeli preveriti vseh 5^{20} možnih načinov izvajanja programa in bi za en testni primer potrebovali le stotinko sekunde, bi to še vedno trajalo več kot 30.000 let. Kaj lahko torej storimo? Čeprav s testiranjem ne moremo dokazati, da napak ni, poskušamo vseeno povečati naše zaupanje v programsko opremo. Na osnovi izkušenj so se izoblikovale metode testiranja, ki preverjajo delovanje programske opreme na najbolj kritičnih mestih in na mestih, kjer se napake najpogosteje pojavljajo. Najmanj, kar lahko naredimo, je, da preverimo delovanje vsake posamezne instrukcije v programski kodi. Obnašanje celotnega sistema tudi skušamo preveriti v vseh možnih realnih pogojih delovanja. S testiranjem skušamo povečati naše zaupanje v programsko opremo in upamo, da bomo s temi testi odkrili vsaj večino napak.

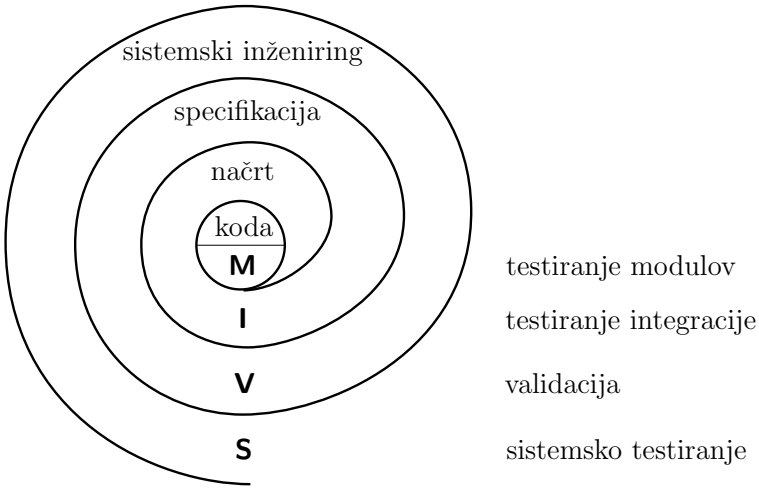
Za razliko od ostalih aktivnosti pri razvoju programske opreme je testiranje s psihološkega vidika nekakšno *destruktivno* delo, saj s testiranjem skušamo "zlomiti" sistem, pokazati, da je z njim nekaj narobe, da ima na-



Slika 10.3: Diagram poteka preprostega programa z eno zanko in petimi možnimi potmi skozi zanko. Za kodiranje takega programa zadostuje že okoli 100 vrstic v Pascalu.

pake ali da ne ustreza zahtevam. Testiranje na določen način tudi preverja delo, ki je bilo opravljeno med analizo, načrtovanjem in kodiranjem. Zato so analitiki, načrtovalci in koderji lahko osebno prizadeti, če se v rezultatih njihovega dela odkrijejo pomanjkljivosti. Če testiranje izvajajo isti ljudje kot prej naštete "konstruktivne" aktivnosti, lahko pride do konflikta interesov. Izberejo "lahke" testne primere, ki ne preverijo delovanje programske opreme z vseh možnih vidikov.

10.1 Metode testiranja



Slika 10.4: Strategija testiranja programske opreme

Testirati začnemo na nivoju posameznih modulov in postopoma napredujemo k integraciji celotnega sistema (slika 10.4). Kot referenca za testiranje med integracijo modulov služi načrt programske opreme. Nato validacija na osnovi zahtev, zapisanih v specifikaciji, ugotavlja, ali jih sistem izpolnjuje. Končno sistemsko testiranje v simuliranih ali realnih razmerah preverja, če razvita programska oprema pravilno deluje skupaj z drugimi sistemskimi sklopi. Za testiranje posameznega modula je v prvi vrsti zadolžen razvijalec modula. Pri velikih sistemih pa kasnejše testiranje izvaja posebna neodvisna skupina za testiranje.

Metode testiranja delimo na dve osnovni skupini:

Metode bele skrinjice ali **strukturna analiza**. Metode v tej skupini pri načrtovanju testnih primerov upoštevajo notranjo strukturo kode.

Metode črne skrinjice ali **funkcionalna analiza**. Pri teh metodah testiranja je notranja struktura kode zastrta. Izbiramo lahko različne vhodne podatke in rezultirajoče izhodne podatke primerjamo s pričakovanimi.

Druga možna delitev metod testiranja je na: *statične metode* (ne zahtevajo izvajanja programa) in *dinamične metode*. Pri statičnih metodah gre za branje dokumentov oziroma sledenje programski kodi — zato lahko govorimo

tudi o ročnih metodah testiranja. Pri dinamičnih metodah pa izvajamo program in njegove rezultate primerjamo s pričakovanimi rezultati.

10.1.1 Strukturna analiza

Po principu bele skrinjice testiramo predvsem posamezne module. Princip bele skrinjice pomeni, da imamo na vpogled notranjo strukturo modula. Na tej osnovi tudi načrtujemo testne primere. Med metode bele skrinjice sodita:

1. *Testiranje glavnih poti.* Osnovni namen metode je, da se vsi programski ukazi izvedejo vsaj enkrat.

2. *Testiranje zank.* Izkušnje kažejo, da pri kodiranju zank še posebej pogosto naredimo napake.

Testiranje glavnih poti

Namen testiranja glavnih poti je, da se vsi ukazi izvedejo vsaj enkrat, ali z drugimi besedami, da s testnimi primeri pokrijemo vso kodo. Ugotoviti moramo torej, koliko neodvisnih poti vodi skozi kodo, in pripraviti za vsako pot testni primer. Število glavnih poti skozi program lahko ugotovimo s pomočjo transformacije diagrama poteka (slika 10.5) v graf poteka (slika 10.6). Graf poteka je usmerjen graf, ki ga sestavljajo vozlišča in robovi, in na bolj pregleden način predstavlja topološko strukturo programa. Vsako vozlišče predstavlja enega ali več zaporednih ukazov. V vozliščih se tok procesiranja loči ali združi. Vozlišča, kjer se potek procesiranja loči, so predikatna vozlišča. Robovi med vozlišči predstavljajo tok procesiranja kot v diagramih poteka. Vozlišča in povezave določajo v grafu regije; ena regija je tudi okolica grafa (slika 10.6).

Neodvisna pot skozi graf poteka je vsaka pot skozi graf, ki vključi vsaj eno novo povezavo. Na primer na sliki 10.6 so neodvisne poti naslednje:

1. pot: 1–16

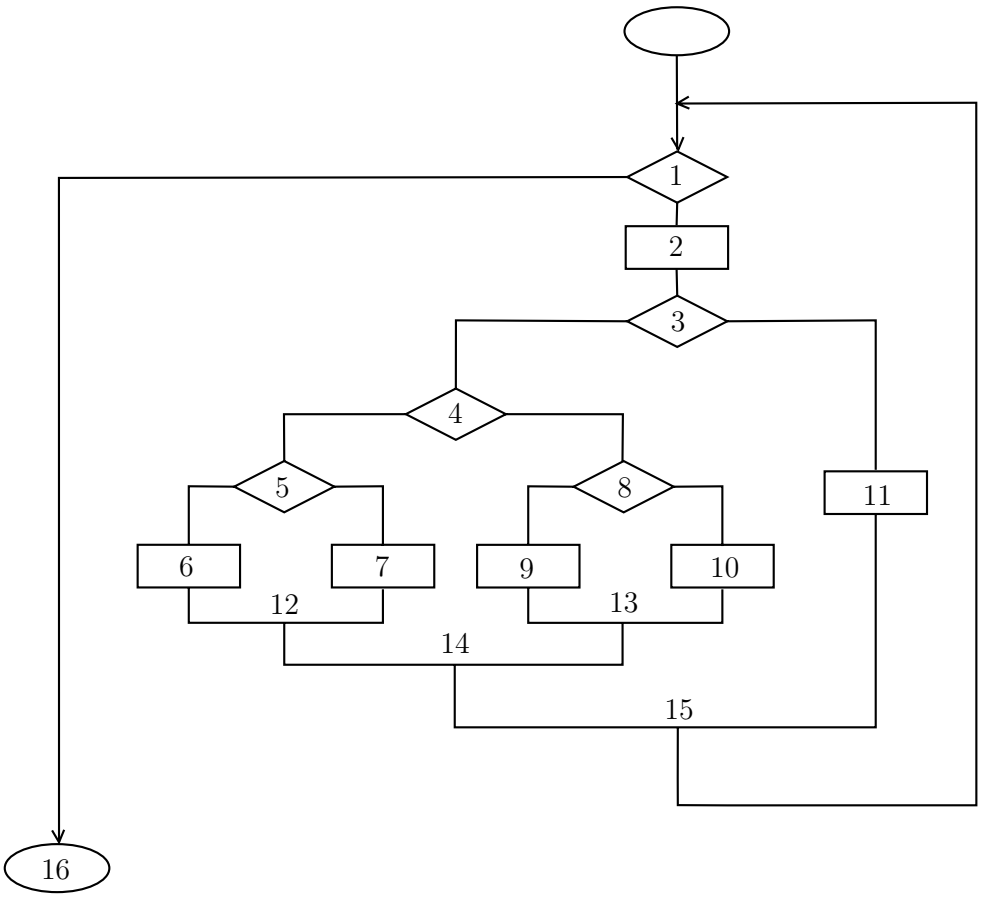
2. pot: 1–2–3–4–5–6–12–14–15–1–16

3. pot: 1–2–3–4–5–7–12–14–15–1–16

4. pot: 1–2–3–4–8–9–13–14–15–1–16

5. pot: 1–2–3–4–8–10–13–14–15–1–16

6. pot: 1–2–3–11–15–1–16



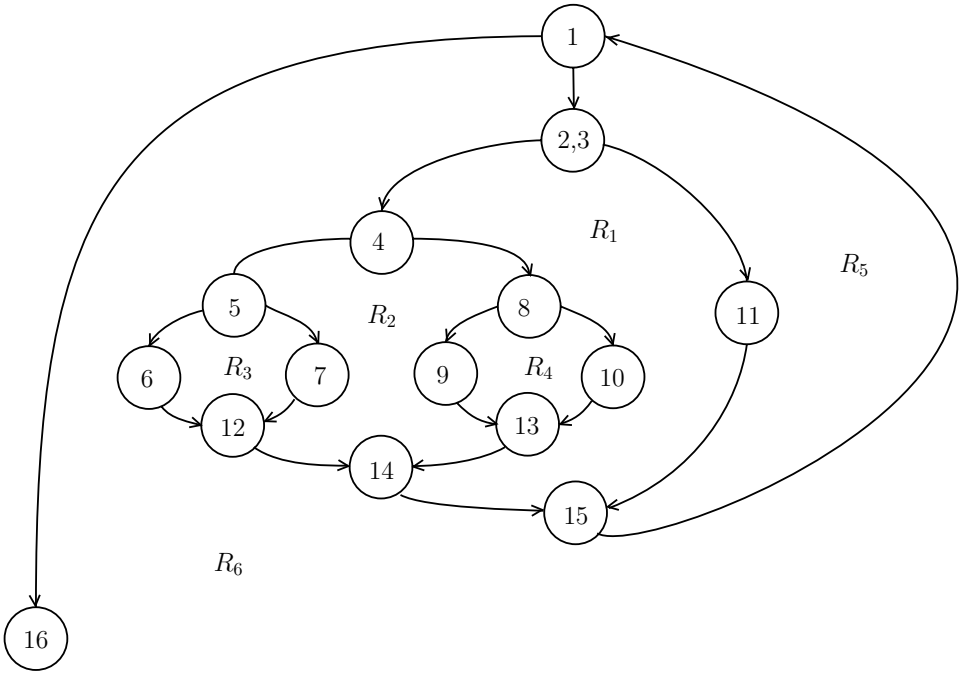
Slika 10.5: Diagram poteka

Število neodvisnih poti skozi graf v teoriji grafov imenujemo ciklomatična kompleksnost $V(G)$. Izračunamo jo lahko na tri načine:

1. $V(G) =$ število regij.
2. $V(G) = E - N + 2$, kjer je E število robov, N pa število vozlišč.
3. $V(G) = P + 1$, kjer je P število predikatnih vozlišč.

Za primer grafa poteka s slike 10.6 so izračuni naslednji:

1. Graf poteka ima šest regij.
2. $V(G) = 19$ robov -15 vozlišč $+2 = 6$



Slika 10.6: Graf poteka (angl. flow graph) za diagram poteka na sliki 10.5. Vozlišča in povezave določajo šest regij.

3. $V(G) = 5$ predikatnih vozlišč $+1 = 6$

Ciklomatična kompleksnost je zgornja meja za število neodvisnih poti in s tem števila testnih primerov, ki jih moramo pripraviti, da bi pokrili vse ukaze v programu. Testni primeri morajo prisiliti program, da ubere izbrano pot procesiranja.

Testiranje zank

Skoraj vsak program oziroma algoritem vsebuje zanke. Pri pisanju zank lahko naredimo drobne, težko opazne napake, še posebej pri minimalnih in maksimalnih možnih ponovitvah. Zato velja sestaviti posebne testne primere za testiranje zank.

Preproste zanke. n je maksimalno število dovoljenih prehodov zanke.

1. Preskoči zanko v celoti.
2. Pojdi skozi zanko enkrat.

3. Pojdi skozi zanko dvakrat.
4. Pojdi skozi zanko m-krat, če je $m < n$.
5. Pojdi skozi zanko $n - 1$, n in $n + 1$-krat.

Vgnezdene zanke. Če bi z načinom testiranja preprostih zank nadaljevali pri vgnezdenih zankah, bi število testnih primerov geometrijsko naraščalo. Zato lahko število testnih primerov zmanjšamo na naslednji način:

1. Začni testiranje pri notranji zanki. Vse ostale zanke postavi na minimalne vrednosti.
2. Testiraj notranjo zanko kot preprosto zanko.
3. Nadaljuj s testiranjem prve naslednje zunanje zanke. Vgnezdene zanke postavi na tipične vrednosti, zunanje pa na minimalno vrednost.
4. Nadaljuj, dokler niso testirane vse zanke.

10.1.2 Funkcionalna analiza

Testiranje po principu črne skrinjice uporabljamo v kasnejših fazah testiranja za preverjanje funkcionalnih zahtev. Zato funkcionalno testiranje ni nadomestilo za strukturno testiranje, saj odkriva druge vrste napak:

- napačne ali manjkajoče funkcije,
- napake vmesnika,
- napake pri branju podatkov v podatkovnih bazah,
- nizko zmogljivost,
- napake pri inicializaciji in na koncu procesiranja.

Pri načrtovanju testnih primerov skušamo izbrati takšne testne primere, da bi lahko s čim manjšim številom testov odkrili čimveč možnih napak. Z enim testnim primerom ne želimo preveriti le posamezne možne napake temveč, če je možno, cel razred podobnih napak. Metodi testiranja po načelu črne skrinjice sta metoda ekvivalentne particije in analiza mejnih vrednosti.

Ekvivalentne particije. Da bi zmanjšali število testnih primerov, vhodne podatke razdelimo na *razrede* ali *ekvivalentne particije*. V idealnem primeru bi za vsak razred zadoščal le en testni primer. Ekvivalentna

particija predstavlja množico veljavnih ali neveljavnih vhodnih podatkov. Običajno so vhodni podatki določena numerična vrednost, interval vrednosti, množica možnih vrednosti ali Boolova spremenljivka. Ekvivalentne vhodne particije so v tem primeru naslednje:

1. Če mora biti vhodni podatek znotraj intervala, definiramo tri ekvivalenčne razrede, enega pravilnega in dva nepravilna.
2. Če na vhodu zahtevamo specifično vrednost, definiramo tri ekvivalenčne razrede, enega pravilnega in dva nepravilna.
3. Če na vhodu pričakujemo člana množice, definiramo dva ekvivalenčna razreda, enega pravilnega in enega nepravilnega.
4. Če na vhodu pričakujemo Boolovo vrednost, definiramo en pravilni in en nepravilni razred.

Analiza mejnih vrednosti. Programerji so opazili, da je običajno več napak pri mejnih vrednostih intervalov vhodnih podatkov kot pa na sredini teh intervalov. Analiza mejnih vrednosti zato omogoča definiranje testnih primerov, ki preverjajo mejne vrednosti dovoljenih podatkov. Navodila za pripravo testnih primerov so podobni kot pri analizi ekvivalentnih particij.

1. Če na vhodu pričakujemo vrednosti iz intervala, ki je omejen z vrednostima a in b, moramo pripraviti testne primere za vrednosti a in b ter vrednosti tik nad in tik pod tema dvema vrednostima.
2. Če na vhodu pričakujemo določeno število podatkov, moramo testirati njihovo minimalno in maksimalno vrednost. Preveriti moramo delovanje programa tudi za vrednosti tik pod in tik nad tema dvema omejitvama.
3. Ista merila uporabimo tudi za izhodne podatke.
4. Če imajo notranje podatkovne strukture omejitve glede števila elementov, je potrebno te omejitve testirati na podoben način.

10.1.3 Ročne metode testiranja

Med ročne metode testiranja, ki ne zahtevajo izvajanja programa na računalniku, sodijo:

Branje. Najbolj tradicionalna metoda preverjanja delovanja programa je branje kode. Vsak, ki programira, nekajkrat prebere kodo, ki jo razvija. Zaradi tega avtor programa postane skoraj slep za nekatere

pomanjkljivosti ali napake, saj bolj misli na to, kako naj program dela. Boljše rezultate dosežemo, če kodo bere kdo drug kot in ne njen avtor.

Drugi razlog, zakaj je branje samega avtorja manj uspešno pri odkrivanju napak, je manjša kritičnost do lastnega dela. Najbolj neodvisno oceno dobimo z anonimno recenzijo (angl. peer review).

Sledenje in preverjanje. Preverjanje programske opreme izvajamo v obliki formalnih recenzijskih sestankov. Na sestanku se kodo prebira in hkrati razlaga njen pomen. Ta interpretacija kode lahko pomaga pri odkrivanju napak.

Sledenje (angl. walktrough) je preverjanje programske kode s pomočjo testnih podatkov. Ker je metoda ročna, morajo biti testni primeri preprosti, saj postane sledenje hitro prezahtevno.

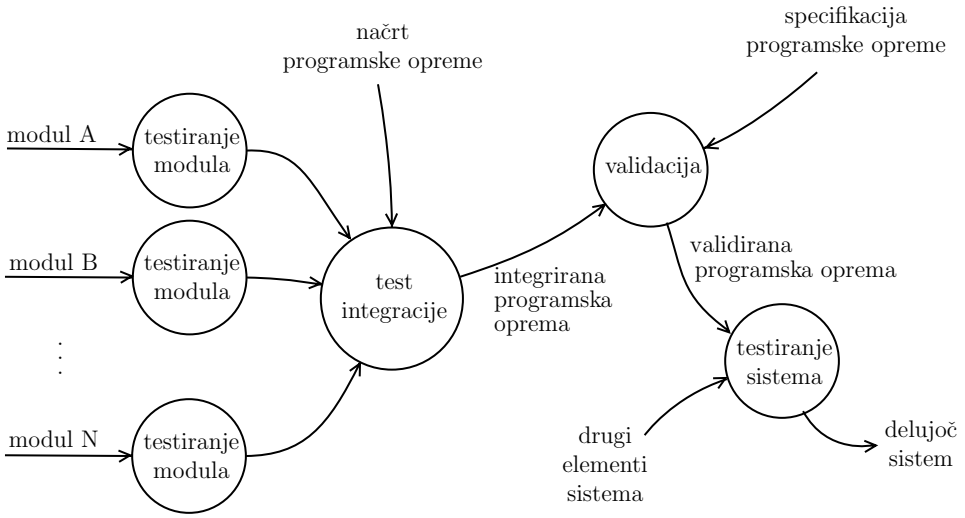
Postopna abstrakcija. Pri načrtovanju programske opreme pogosto uporabljamo postopno izboljšavo in s tem nižamo abstraktno raven obravnave problema. Metoda postopne abstrakcije gre v nasprotno smer. Iz programske kode skušamo v nekaj korakih razbrati funkcijo, ki jo ta koda implementira. Funkcijo, ki jo na ta način abstrahiramo, primerjamo s funkcijo, opisano v specifikaciji ali načrtu za ta del programske kode. Obe funkciji bi seveda morali biti skladni.

10.1.4 Dokazovanje pravilnosti

Pri formalnem dokazovanju pravilnosti programov skušamo dokazati, da program ustreza svoji specifikaciji. Zato mora biti tudi specifikacija zapisana na formalen način. Specifikacijo formalno zapišemo običajno tako, da napišemo trditev, ki velja pred izvajanjem programa in trditev, ki mora veljati po izvajanju programa. Dokazati moramo, da program prevede prvo trditev v drugo. Uporabljamo lahko matematično indukcijo ali predikatni račun. Dokazovanje pravilnosti na tak formalen način je izvedljivo le za kratke in relativno preproste programe. Primer programov, kjer se formalno dokazovanje uporablja, so razni komunikacijski protokoli. To so relativno kratki programi, ki se izredno velikokrat izvajajo. Za velike in kompleksne programe pa dokazovanje pravilnosti žal še ni praktično izvedljivo.

10.2 Postopek testiranja

Testiranje začnemo pri posameznih modulih, ki jih testiramo po principu bele skrinjice. Testirane module nato postopoma združujemo ali integri-



Slika 10.7: Testiranje programske opreme

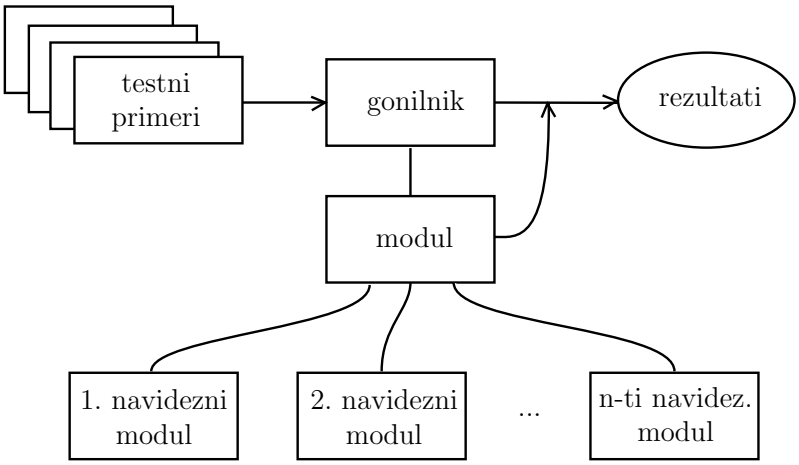
ramo. Ko je programska oprema sestavljena, preverimo še, če ustreza zahtevam, zapisanim v specifikaciji. Nazadnje opravimo še razna sistemska testiranja (slika 10.7).

10.2.1 Testiranje modulov

Posamezne module testiramo po načelu bele skrinjice. S testiranjem moramo preveriti vse poti skozi modul. Po potrebi testiramo tudi zanke, vmesnike in druge elemente. Za testiranje posameznih modulov moramo sestaviti posebno testno konfiguracijo, ki nadomesti sistem, v katerega bo modul kasneje vgrajen (slika 10.8). Potreben je gonilnik (angl. driver), ki kliče testirani modul, in navidezni moduli (angl. stub), ki nadomeščajo podrejene module. Bolj ko je modul notranje enoten in manj ko je soodvisen, lažje ga je testirati.

10.2.2 Testiranje integracije

Pri povezovanju sicer pravilnih modulov lahko pride do številnih napak zaradi neusklajenih vmesnikov in skupnih podatkovnih struktur. Testiranje med integracijo poteka po principu črne skrinjice — pripravimo vhodne podatke in nato primerjamo dejanske izhodne podatke s pričakovanimi. Integracija mora potekati postopno. To pomeni, da vsakokrat, ko sistem



Slika 10.8: Testiranje modulov

razširimo za en modul, zopet ponovimo testiranje. To imenujemo regresijsko testiranje. Če bi hkrati združili več modulov ali celo sestavili celoten sistem, bi le težko izolirali vzroke za napake.

Integracija lahko poteka od zgoraj navzdol ali od spodaj navzgor. Katero pot je smiselno ubrati, je odvisno od težavnosti testiranja. Med testiranjem moramo namreč module, ki še niso integrirani v sistem, nadomestiti z navideznimi moduli.

Integracija od zgoraj navzdol. Začnemo z glavnim ali kontrolnim modulom. Postopoma dodajamo podrejene module, v globino ali v širino. Če integriramo najprej v globino, lahko prej demonstriramo določeno funkcijo sistema. Potrebni so le začasni podrejeni moduli. Problemi lahko nastopijo le, če je simuliranje procesiranja v nižje ležečih modulih težavno. Po kompleksnosti razlikujemo štiri vrste navideznih modulov:

- prikažejo vedno enako sporočilo,
- prikažejo posredovano sporočilo,
- vrniti morajo vrednost iz tabele,
- glede na posredovani parameter morajo poiskati vrednost v tabeli in jo vrniti nadrejenemu modulu.

Integracija od spodaj navzgor poteka tako, da module na nižjem nivoju združimo v skupine. Napisati moramo gonilnike, ki koordinirajo testiranje skupin modulov. Postopoma odstranjujemo gonilnike, jih

nadomeščamo s pravimi moduli ter skupine združujemo. Podobno kot pri navideznih modulih tudi pri gonilnikih lahko po kompleksnosti ločimo štiri vrste gonilnikov.

Da bi zmanjšali število potrebnih navideznih modulov, ali da bi najprej testirali najbolj kritične module, lahko kombiniramo oba načina sistemske integracije.

10.2.3 Sistemsko testiranje

Končna faza testiranja nastopi, ko je sistem že integriran. Sistem naj bi takrat že pravilno deloval, preveriti pa moramo, ali sistem res deluje tako, kot je predpisano v specifikaciji — to je *validacija* sistema.

Ko je programska oprema že instalirana v svoje delovno okolje, je potrebno testirati še celoten sistem. Zanima nas hitrost programske opreme, zmožnost okrevanja sistema po izpadu sistema, maksimalna obremenitev sistema, varnost in občutljivost sistema, kaj se zgodi pri vnašanju napačnih podatkov in ukazov ter podobno.

Pri *testiranju okrevanja sistema* programsko opremo na različne načine prisilimo, da izpade. Nato opazujemo, če sistem pravilno in hitro okreva, tako kot od njega zahtevamo, bodisi avtomatično bodisi z operatorjevim posredovanjem.

Pri *testiranje varnosti* moramo predvsem ugotoviti, če sistem preprečuje neavtoriziran dostop.

Pri *testiranju obremenitve* sistem izpostavimo nenormalnim obremenitvam (število uporabnikov, število vhodnih podatkov, poraba spomina itd.) in opazujemo, kdaj sistem izpade ali postane neuporaben.

Pri *testiranju občutljivosti* skušamo odkriti, ali lahko določene kombinacije vhodnih podatkov, ki so posamezno obravnavani sicer pravilni, povzročijo nestabilnost sistema oziroma napačno procesiranje.

Programsko opremo, namenjeno širšemu krogu uporabnikov, pred prodajo pogosto damo v poskusno uporabo zainteresiranim posameznikom ali skupinam v obliki tako imenovamih verzij α in β.

10.3 Razhroščevanje

Razhroščevanja (angl. debugging) ali popravljanja napak ne smemo zamenjevati s testiranjem. Testiranje pokaže na napake, razhroščevanje pa mora poiskati njihove vzroke in jih odpraviti.

Iskanje vzrokov za napake pa je težavno:

1. Vzrok in posledica napake sta lahko v kodi zelo oddaljena in različna.
2. Napaka lahko (začasno) izgine, ko popravimo drugo napako.
3. Vzrok napake je lahko zaokroževanje in ne napačno procesiranje.
4. Vzrok za napako je lahko le v določenem časovnem zaporedju operacij.
5. Vzrok je lahko v napačnih predpostavkah specifikacije sistema.

Pri iskanju napak igra veliko vlogo osebna nadarjenost in neke vrste intuicija, ki ni odvisna le od izkušenj. Pri iskanju napak si pomagamo s povratnim sledenjem po kodi od točke, ko napako zaznamo. Kadar je več možnih vzrokov za napako, jih po vrsti preverjamo in izločamo.

10.4 Formalna tehnična recenzija

Postopek formalnih tehničnih recenzij je eden od najvažnejših mehanizmov za zagotavljanje kvalitete. Njeni cilji so:

- odkrivanje napake v funkcijah, logiki in implementaciji katerekoli faze v razvoju programske opreme,
- validacija zahtev,
- usklajevanje s standardi,
- uniformen razvoj programske opreme,
- lažje upravljanje s projektom razvoja programske opreme.

Formalno tehnično recenzijo izvajamo v obliki recenzijskih sestankov. Na takih sestankih naj bi sodelovalo od tri do pet ljudi. Priprave na sestanke naj ne bi bile predolge. Tudi trajanje sestankov naj bi bilo omejeno. Zato je bolje imeti več ozko usmerjenih sestankov. Zaključki sestankov morajo biti povsem jasni. Sklepi recenzijskega sestanka so lahko naslednji:

- recenzirani produkt se sprejme brez sprememb,
- produkt se zaradi napak zavrne,
- produkt se pogojne sprejme (popraviti je treba drobne napake, ponovna recenzija pa ni potrebna).

Za delovanje recenzijskih skupin je potrebno sprejeti pravila o njihovem delovanju. Recenzirati je potrebno produkt, ne pa njenih avtorjev. Recenzija naj zato ne deluje kot inkvizicija, temveč konstruktivno. Recenzijske skupine naj bodo sestavljene iz različno izkušenih ljudi. Mlajši člani skupine se pri recenziranju lahko veliko načijo od starejših, bolj izkušenih razvijalcev.

Poglavje 11

Vzdrževanje programske opreme

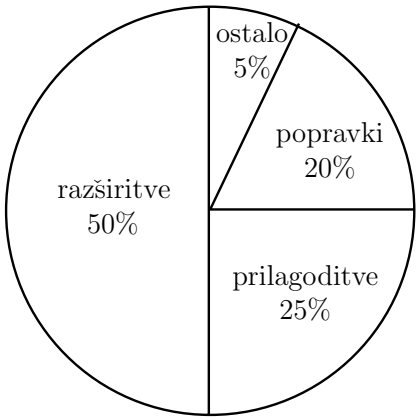
Vzdrževanje programske opreme je bilo do nedavnega najbolj zanemarjena aktivnost v življenjskem ciklu programske opreme. Stroški vzdrževanja programske opreme stalno naraščajo, saj je na svetu vedno več programske opreme. Tehnično zastarele programske opreme je enostavno toliko, da je ne moremo dovolj hitro nadomeščati z novo. Obstoj različnih organizacij je povsem odvisen od nemotenega delovanja programske opreme. Čeprav je programska oprema tehnično zastarela, jo je potrebno nenehno prilagajati različnim zunanjim spremembam, od sprememb v zakonodaji, do širitve poslovanja. Informacijski sistemi se starajo, njihovo vdrževanje pa stane vedno več (tabela 11.1).

Tabela 11.1: Pomembnost vzdrževanja programske opreme se zrcali tudi v naraščajočem deležu v skupni ceni programske opreme.

Delež vzdrževanja	1970-ta leta	1980-ta leta	1990-ta leta
v ceni programske opreme:	35–40%	40–60%	70–80%

Po nekaterih ocenah je kar 80% programske opreme, ki je v uporabi po vsem svetu, nestrukturirane in slabo dokumentirane. Vendar kljub boljši kvaliteti novo nastajajoče programske opreme stroški vzdrževanja zaradi večjega obsega in večje kompleksnosti programske opreme ostajajo enaki ali se še višajo. Vzdrževanje programske opreme poraja tudi indirektne stroške,

saj težave pri vzdrževanju povzročajo nezadovoljstvo strank in zavlečejo razvoj nove programske opreme.



Slika 11.1: Delež različnih aktivnosti pri vzdrževanju programske opreme

Vzdrževanje programske opreme obsega naslednje štiri glavne aktivnosti:

1. **Popravki za odpravljanje napak.**
2. **Prilagajanje programske opreme** spremembam v okolju, na drugo vrsto strojne ali sistemske programske opreme. Pri tem se funkcionalnost programske opreme ne spremeni.
3. **Razširitev zmogljivosti programske opreme** se ukvarja z novimi ali spremenjenimi zahtevami. Spremeniti moramo funkcije sistema, izboljšamo lahko zmogljivost sistema ali uporabniški vmesnik.
4. **Preventivno vzdrževanje programske opreme** je namenjeno spremembam, ki izboljšajo možnost vzdrževanja (dopolnjevanje dokumentacije, izboljšava programske strukture itd.).

Delež odpravljanja napak med vsemi vzdrževalnimi aktivnostmi znaša le okoli 25% (slika 11.1). Največ zahtev za vzdrževanje programske opreme je rezultat različnih sprememb v zunanjem okolju.

Večina težav pri vzdrževanju je posledica nedoslednosti v razvoju programske opreme, nestrukturirane kode, nepopolne specifikacije in slabe dokumentacije. Vrh vsega vzdrževanje pri programerjih ni priljubljeno delo, saj večina programerjev raje piše nove programe kot popravlja stare. Čim kompleksnejša je programska oprema, tem težje se jo vzdržuje in tem večji problemi nastajajo pri iskanju njenih vzdrževalcev. Obsežne projekte razvoja

Anegdota o popravljanju programske opreme po pripovedi Davida Parnasa

Programsko opremo za kontrolo upravljanja vojaškega letala je bilo potrebno napisati na novo. Letalo je imelo dva višinomera. Programska oprema je po vrsti poskušala prebrati podatke dveh ločenih višinomerov in prikazati rezultat na komandni plošči. Originalna programska oprema je bila naslednja:

```
IF not-read1 (V1) GOTO DEF1;
display (V1);
GOTO C;
DEF1:  IF not-read (V2) GOTO DEF2;
display (V2);
GOTO C;
DEF2:  display (3000);
C:
```

Vzdrževalec je kodo najprej spremenil v strukturirano obliko:

```
if read-meter1 (V1) then display (V1) else
if read-meter2 (V2) then display (V2) else
   display (3000)
endif
```

Zakaj je sistem prikazal vrednost 3000, če ni mogel prebrati podatkov nobenega od obeh višinomerov, vzdrževalcu ni bilo jasno. O tem ni bilo nobene dokumentacije. Končno mu je uspelo govoriti s prvotnim programerjem, ki mu je pojasnil svojo odločitev. Ker ni vedel, kaj naj prikaže na zaslonu, če nobeden od obeh višinomerov ne dela, je vprašal enega od pilotov, na kateri višini se v povprečju leti s to vrsto letal. Pilot je ocenil, da je povprečna višina letenja 3000 čevljev!

Vzdrževalec je pravilno zaključil, da programska oprema ne bi smela okvare višinomerov obravnavati na tak način. Lovska letala letijo zelo visoko ali zelo nizko in le redko nekje vmes. Vzdrževalec je zato zaprosil nadrejene za dovoljenje, da v tem primeru na zaslonu prikaže opozorilo "DVIGNI SE" (angl. "PULL UP"). Toda to so nadrejeni zavrnili! Cele generacije pilotov so se namreč dotedaj že naučile pravilno reagirati na številko 3000. Celo v učnih priročnikih je pisalo "Če je prikazana višina 3000 dlje kot sekundo na zaslonu, se takoj dvigni!"

programske opreme, ki trajajo nekaj let, ne obvladajo dobro niti njeni ustvarjalci, še manj drugi vzdrževalci.

Da bi zmanjšali stroške vzdrževanja, moramo upoštevati deleže aktivnosti v tabeli 11.1. Prilagoditvam programske opreme na spremembe v okolju se ni možno izogniti, saj je programska oprema del nenehno se spreminjajoče realnosti. Lahko pa z večjo kvaliteto programske opreme zmanjšamo število napak in pri njenem načrtovanju predvidimo, katere dele programske opreme bo v prihodnosti največkrat potrebno spreminjati. Objektno orientirano načrtovanje omogoča lažjo izolacijo tistih delov sistema, ki se pogosto spreminjajo. Stroški vzdrževanja so tudi manjši, če je obseg programske kode manjši.

Čeprav je programska oprema dobro načrtovana in zgrajena, jo je s časom vse težje vzdrževati. Dva zakona evolucije programske opreme (glej 2. poglavje) sta še posebej pomembna za vzdrževanje programske opreme:

Zakon stalnih sprememb. Sistem je smiselno vzdrževati toliko časa, dokler ga ni ekonomsko bolj upravičeno nadomestiti z novim sistemom.

Zakon naraščajoče kompleksnosti. Zaradi sprememb se programskim sistemom slabša struktura (entropija narašča) in zato postajajo vse bolj kompleksni. Za preprečevanje pretiranega povečanja kompleksnosti programske opreme je potrebno dodatno delo.

Da bi vzdrževalci lahko spreminjali programsko opremo, jo morajo najprej razumeti. Spremljajoče dokumentacije pa pogosto sploh ni ali pa je zastarela. Pogosto se pri spreminjanju in popravljanju programske opreme tako mudi, da se ne naredi ustreznih sprememb v dokumentaciji. Dokumentacija tudi običajno opisuje le končni rezultat oziroma kodo. Za lažje razumevanje pa bi morali vedeti tudi, zakaj je bila neka rešitev izbrana med vsemi drugimi možnimi rešitvami. V dokumentaciji so zelo koristni testni primeri, ki olajšajo preverjanje programske opreme po vzdrževalnem posegu.

Na zmožnost vzdrževanje programske opreme vpliva tudi uporaba bolj razširjenih programskih jezikov, strojnih in sistemskih platform.

Ker vzdrževanje programske opreme ni tako ugledno kot razvoj nove in ker je to delo pogosto slabše plačano, vzdrževalci niso zadovoljni in pogosto menjavajo svojo zaposlitev, kar še dodatno poslabša možnost kvalitetnega vzdrževanja. Vzdrževanje programske opreme, ki jo nismo napisali sami, je v resnici zelo zahtevno delo. Zahteva še več izkušenj kot pisanje nove programske opreme. Tudi na nemoteno tekoče delovanje organizacij ima vzdrževanje obstoječe programske opreme večji vpliv kot razvoj nove.

Z vzdrževanjem programske opreme je povezan pojem "reverznega inženistva" (angl. reverse engineering). Cilj reverznega inženirstva je na osnovi analize obstoječega sistema identificirati njegove komponente in njegovo strukturo. Zgraditi mora nov, funkcionalno enak ali boljši sistem v drugačni obliki ali na višji stopnji abstrakcije.

11.1 Organizacija vzdrževanja programske opreme

Vzdrževanje programske opreme moramo primerno organizirati. Skupine za razvoj in vzdrževanje programske opreme lahko organiziramo po treh načelih [63]:

W-type: delitev po tipu dela (angl. work), na primer analiza in programiranje.

A-type: delitev glede na vrsto aplikacije.

L-type: delitev glede na življenjski cikel (angl. life-cycle), na primer razvoj programske opreme in vzdrževanje programske opreme.

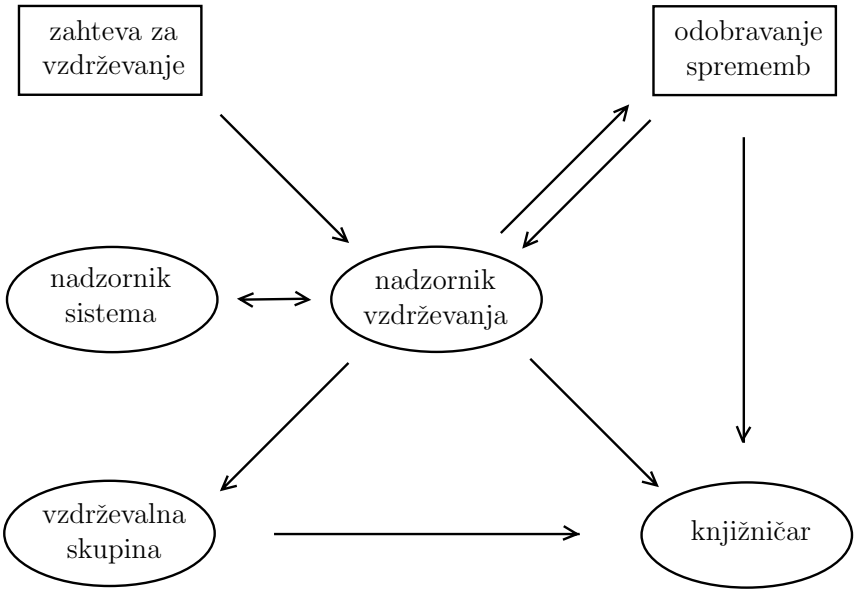
Delitev razvoja in vzdrževanja programske opreme ima prednosti in slabosti. Prednosti ločenega razvoja in vzdrževanja so [63]:

- jasne zadolžitve; če isti ljudje razvijajo in vzdržujejo programsko opremo, je težko nadzorovati, koliko časa kaj delajo;
- zaradi nujnih vzdrževalnih del je težko časovno načrtovati razvoj nove programske opreme;
- delitev razvoja in vzdrževanja postavlja jasno ločnico med obe aktivnosti; s to ločnico so običajno povezani zaključni testi;
- s specializacijo aktivnosti je možna višja kvaliteta;
- z bolj osredotočenim delom je možna večja produktivnost.

Slabosti ločenega razvoja in vzdrževanja programske opreme so [63]:

- slabša motivacija ljudi zaradi različnega statusa dela, ki ga opravljajo;
- slabše poznavanje sistema (načrta in aplikacije);
- slabša koordinacija med razvojem in vzdrževanjem, še posebej ko gre za razvoj novega sistema, ki bo nadomestil starega;

- večji začetni stroški vzdrževanja;
- možno podvajanje komunikacij z uporabnikom.



Slika 11.2: Za vzdrževanje je potrebno vzpostaviti ustrezno koordinacijo dela.

11.1.1 Postopek vzdrževanja

1. Vzdrževanje programske opreme moramo ustrezno organizirati, da ne bi prišlo do neavtoriziranih ali nasprotujočih sprememb programske opreme (slika 11.2).
2. Vsak zahtevek za vzdrževanje mora biti skrbno dokumentiran. V primeru napake moramo dokumentirati okoliščine, ki so pripeljale do napake.
3. Zahtevke moramo preveriti in odobriti le tiste, ki so potrebni, smiselni in ekonomsko upravičeni.
4. Glede na vrsto vzdrževanja sprožimo ustrezen postopek. V primeru kritične napake, ki zaustavi normalno delovanje neke organizacije, moramo takoj ukrepati z vsemi razpoložljivimi silami (slika 11.4). Če gre

Slika 11.3: Kvalitetno vzdrževanje programske kode je možno, če začnemo s spremembami na najvišjem nivoju. Spremembe nato postopoma prenašamo po vsej hierarhiji dokumentov do same programske kode.

za zadevo, ki še lahko počaka, jo uvrstimo na seznam napak, ki jih rešujemo po vrsti na ustaljen način.

5. Z organizacijskega vidika obravnavamo vsak zahtevek podobno kot razvoj nove programske opreme. Zahtevo za vzdrževanje moramo skrbno analizirati, spremembo ali popravek načrtovati in na koncu programsko opremo testirati (slika 11.3).

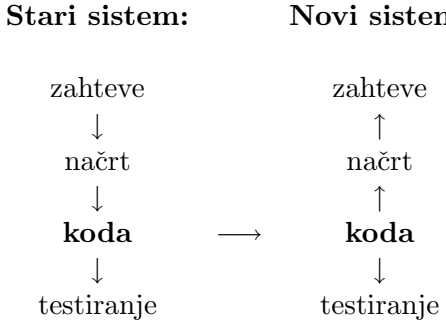
   Tudi za vzdrževanje programske opreme obstaja nekaj modelov za ocenjevanje dela. Model COCOMO podaja naslednjo oceno za delo (v enotah človek – mesec) za vzdrževanje programske opreme v enem letu:

$$E_{\text{maint}} = 2.4 ACT (KLOC)^{1.05} \tag{11.1}$$

$$ACT = \frac{KLOC}{C_1} \tag{11.2}$$

   kjer je $KLOC$ število programskih vrstic obravnavane programske opreme v tisočih, C_1 pa število ukazov, ki smo jih spremenili ali jih dodali v enem letu.

6. Vse spremembe moramo dokumentirati. Tudi vso ostalo dokumentacijo, na primer navodila za uporabo, moramo ustrezno popraviti.



Slika 11.4: Hitre popravke programske opreme delamo običajno tako, da kodo kar spremenimo in jo ponovno prevedemo, da dobimo nov sistem. Ostale dokumente spremenimo šele potem, ko je koda že spremenjena, in če nam to dopušča čas. Žal se prepogosto to nikoli ne zgodi.

11.1.2 Stranski učinki vzdrževanja

Pri vsakem poseganju v obstoječo programsko kodo moramo posebej paziti, da ne povzročimo nepredvidenih stranskih učinkov, kar je še posebej nevarno, če posegamo v programsko kodo brez predhodne analize (slika 11.4). Zato moramo postopati zelo previdno. Posebej nevarni so naslednji posegi:

- spreminjanje ali brisanje imen spremenljivk,
- spreminjanje ali brisanje podprogramov,
- spreminjanje mejnih vrednosti,
- spremembe za pohitritev procesiranja.

Poglavje 12

Smeri razvoja programske opreme v prihodnosti

Procesorska moč, spominske in komunikacijske kapacitete se večajo z nezmanjšano hitrostjo, kar dobro opisuje Moorov zakon; načrtovanje in implementacija kompleksne programske opreme pa ostaja draga in z napakami obremenjena dejavnost.

Čeprav se tudi na področju programskega inženirstva stalno porajajo nove ideje, metode in orodja, ostaja razvoj programske opreme težaven zaradi kompleksnosti same programske opreme. Ko kompleksnost neke domene zelo naraste, eksperti težko formulirajo svoje znanje o tej domeni. Ta fenomen je na področju ekspertnih sistemov znan kot Feingenbaumovo ozko grlo (angl. Feigenbaum bottleneck). Človeški eksperti veliko lažje demostrirajo svoje znanje tako, da pokažejo, kako probleme rešujejo (angl. "show-how") ali katera rešitev je boljša (angl. "say-what"), kot da bi znali formulirati praktične strategije, kako se naj problemi rešujejo (angl. "say-how"). Že v pionirskih časih programskega inženirstva so se izoblikovala splošna načela, kakšen naj bo dober sistem (glej str.142). Zaradi Feingenbaumovega ozkega grla pa je izredno težko natančno opisati korake, po katerih bi lahko zgradili sisteme z želenimi lastnostmi [50]. Želene lastnosti sistemov veliko lažje opišemo s splošnimi načrtovalskimi načeli. Teh načel se je na osnovi bogatih izkušenj nabralo že zelo veliko. Cela vrsta knjig in člankov je namenjena tej problematiki. Kljub velikemu obsegu znanja ostaja načrtovanje težavno. Nedoločenost načrtovalskih načel izhaja delno iz njihovega zapisa v naravnem jeziku, delno pa iz želje, da bi z enim načelom opisali veliko število posameznih primerov. Uspešen razvoj programske opreme na osnovi načel je zato v veliki meri odvisen od izkušenj

posameznega razvijalca. Izkušenih programerjev pa je za hitro rastočo industrijo programske opreme vedno premalo. Posebej v velikih industrijskih državah (ZDA, Japonska ipd.) primanjkuje izkušenih programerjev. To je dobra priložnost za podjetja v drugih delih sveta, kjer je na voljo izobražena delovna sila (Indija, vzhodna Evropa). Nekatera slovenska podjetja za razvoj programske opreme so se tudi že uspešno uveljavila na svetovnem trgu programske opreme (npr. Hermes SoftLab, MG-Soft itd.).

V programskem inženirstvu žal tudi izkušnje hitro zastarijo. Razen splošnih načel, organizacijskih in aplikacijskih izkušenj, ki ne zastarijo hitro, se morajo programerji stalno učiti novih metod in se prilagajati na nova orodij ter okolja, s katerimi gradijo svoje izdelke (strojna oprema, operacijski sistemi, komunikacijski protokoli itd.). Nekatera orodja so tako kompleksna, da potrebuje posameznik leto dni ali dlje, da jih res dobro obvlada. Cena razvoja programske opreme je v največji meri odvisna od hitrosti njenega razvoja, saj so človeški viri glavni strošek. Hiter razvoj pa je pomemben tudi z vidika konkurence in obvladovanja tržišča, saj prvi ponudnik neke rešitve lažje najde stranke kot ostali.

Razvoj programske opreme je možno pohitriti in poenostaviti z:

- načrtovanjem in implementacijo na višjem abstraktnem nivoju,
- ponovno uporabo rezultatov.

12.1 Dvigovanje abstraktne ravni reševanja problema

Prve računalnike je bilo potrebno programirati na ravni njihovega strojnega jezika. Z uporabo zbirnega jezika se je raven komunikacije z računalnikom nekoliko dvignila. Pisanje programov v zbirniku je bilo zamudno, poleg tega pa programi še niso bili prenosljivi na druge vrste računalnikov. Šele s pojavom višjih programskih jezikov in njihovih prevajalnikov so programi postali bolj univerzalni in prenosljivi. Želja po definiranju računalniških rešitev na vedno višjem nivoju pa ostaja aktualna, saj želimo načrtovati vedno bolj kompleksne sisteme. Kompleksnost pa lahko obvladujemo le tako, da ustrezno dvigujemo abstraktni nivo obravnave problemov.

Ko osvojimo nov abstraktni nivo obravnave, rešujemo probleme neposredno z uporabo osnovnih operacij in struktur tega nivoja. Z nabiranjem izkušenj začnemo ugotavljati, kateri problemi so podobni in jih lahko rešimo na podoben način. Oblikovati se začnejo komponente, ki predstavljajo standardne rešitve standardnih problemov. Ko je neko problemsko področje

dodobra znano, lahko razvoj aplikacij na tem področju postopoma avtomatiziramo. To pomeni, da moramo neformalni način opisovanje problemov nadomestiti s formalnim jezikom, ki bo s tem omogočil nastanek ustreznega virtualnega računalnika. Formalni jezik mora imeti dovolj izrazne moči za opis problemov. Semantične enote tega novega formalnega jezika postanejo prej opisane standardne komponente.

Računalniško urejanje tekstov

Dober primer problemskega področja, kjer je možno avtomatizirati reševanje problemov s pomočjo bolj abstraktnega formalnega jezika, je urejanje tekstov. Eden od osrednjih problemov pri urejanju tekstov je razdelitev besed v odstavku na posamezne vrstice. Donald E. Knuth je v svojem sistemu TEX problem rešil z definicijo ustreznih semantičnih enot — uporabil je pojme "škatel", "lepila" in "kazni" [30]. Besede je postavil v škatle, ki imajo določeno širino. Presledke med besedami je izrazil s pojmom "lepila", ki se lahko krči ali razteza. Normalni presledek ima kazen "0", skrčeni ali raztegnjeni presledki pa so sorazmerno kaznovani. V posameznem odstavku se kazni seštevajo. Problem razvrstitve besed odstavka v vrstice lahko zdaj formalno izrazimo. Besede v odstavku razvrstimo v vrstice tako, da bo kazen najnižja.

Korak k bolj abstraktni ravni reševanja problemov predstavljajo jeziki 4. generacije, ki so znani tudi kot generatorji aplikacij (angl. application generators). Za razliko od jezikov 3. generacije, lahko z jeziki 4. generacije programiramo na višjem nivoju. Jeziki 4. generacije imajo namreč vgrajene konstrukte, ki so značilni za določeno področje uporabe. Zaradi vgrajenega znanja o področju uporabe so jeziki 4. generacije primerni le za določena in omejena področja uporabe. Podoben cilj imajo raziskovalci, ki skušajo razviti *formalne specifikacijske jezike*, katerih prevajalniki naj bi specifikacijo avtomatično prevedli v višji programski jezik. Taki specifikacijski jeziki naj bi hkrati omogočili odkrivanje napak in nedoslednosti že v fazi analize in načrtovanja in na ta način zagotovili pravilnost sistema. Obsežno testiranje tako razvite kode zato ni potrebno, pri vzdrževanju pa ni potrebno popravljati obstoječe kode, temveč lahko samo ustrezno spremenimo njeno specifikacijo, saj je ves nadaljnji postopek avtomatičen. Ni naključje, da je večina orodij 4. generacije namenjenih poslovnim aplikacijam, saj poslovne aplikacije predstavljajo levji delež vse obstoječe programske opreme.

12.2 Ponovna uporabnost programske opreme

Razvoj programske opreme bi lahko pospešili in poenostavili, če bi lahko segmente programske kode ponovno uporabili in iz takih univerzalnih sestavnih delov zgradili nov sistem. Druga možnost ponovne uporabe je, da obstoječ načrt oziroma sistem prilagodimo novi rabi. Zato lahko ponovno uporabnost na področju razvoja programske opreme dosežemo na dva načina:

- s ponovno uporabo komponent,
- s ponovno uporabo načrta ali sistemske arhitekture.

12.2.1 Ponovna uporaba komponent

Ponovna uporaba programskih komponent je na določenih uporabniških področjih ustaljena praksa. Posebej za matematične in statistične funkcije ali metode obstaja cela vrsta podprogramskih knjižnic. Vendar je tako enostavna ponovna uporaba kode možna le na tistih področjih, kjer obstaja standardna terminologija (npr. funkcija *cos* pomeni vsem isto), standardizirani podatkovni tipi in preprosti vmesniki. Za bolj splošno uporabo programskih komponent pa je prej potrebno rešiti naslednje probleme [65]:

Iskanje. Da bi lahko poiskali primerno komponento v bazi razpoložljivih komponent, jo moramo znati opisati.

Razumevanje. Da bi lahko presodili, če je neka komponenta uporabna, moramo natančno razumeti, kaj komponenta dela.

Adaptacija. Če neka komponenta ni neposredno uporabljiva, jo lahko prilagodimo le, če natančno vemo, kako deluje.

Kompozicija. Sistem je sestavljen iz več komponent. Znati moramo različne komponente sestaviti v enoten sistem.

Module, ki so napisani v objektno orientiranih jezikih, je zaradi odlik objektno orientiranega pristopa še lažje modificirati in na ta način ponovno uporabiti.

Nekoliko več fleksibilnosti dosežemo s tako imenovanimi programskimi vzorci (angl. templates), ki so nekakšne nedokončane komponente. Z dopolnitvijo manjkajočih definicij, večinoma podatkovnih tipov, dobimo običajno komponento. Primer takega vzorca je modul za urejanje z mehurčki, kjer moramo sami določiti število možnih elementov in njihov podatkovni tip. Bolj je modul splošen, več podrobnosti moramo dopolniti. Kadar gre za

vzorce celotnih programov za posebna problemska področja, dobimo generatorje aplikacij. Tipično računalniško področje, kjer se ponovno uporablja celotna sistemska arhitektura, je gradnja prevajalnikov. Objektno orientirane, ponovno uporabne komponente so na voljo predvsem za Visual Basic.

12.2.2 Ponovna uporaba načrta

Kadar moramo rešiti problem, ki je podoben nekemu uspešno rešenemu problemu, je smiselno uporabiti uspešno rešitev že na ravni njenega načrta. Vsak programer pri svojem delu pravzaprav stalno uporablja rešitve, ki so se v preteklosti izkazale kot uspešne. Če povzdignemo ta princip na raven celotne aplikacije, govorimo o načrtovalskih vzorcih (angl. design patterns). Vzorci celotnih ponovno uporabljivih aplikacij, zgrajenih na objektno orientiranih principih, se imenujejo okviri (angl. frameworks) [29]. S specializacijo okvirne aplikacije dobimo aplikacijo, ki je prilagojena našim zahtevam. Okvirna aplikacija je torej nekakšen skelet bodoče aplikacije. Prve okvirne aplikacije so bile namenjene izdelavi grafičnih uporabniških vmesnikov. Ponovno uporabljive komponente in primeren skelet se med seboj dopolnjujejo, saj omogočajo lažje povezovanje komponent. Primeri takih sistemskih komponent so: OLE, OpenDoc in Java Beans. Okviri lahko temeljijo na principu bele ali črne skrinjice. Princip bele skrinjice pomeni, da mora razvijalec poznati notranje delovanje komponent. "Črne skrinjice" je lažje uporabljati, toda težje razviti, saj je potrebno že vnaprej predvideti več možnih načinov uporabe. Standardni okvir za poslovne aplikacije postaja sistem CORBA.

12.2.3 Ekonomska plat ponovne uporabe

Ponovno uporabo programske opreme je možno uvesti le v ustreznih razmerah. Prilagoditi je potrebno vodenje projektov razvoja programske opreme in ustvariti primerno organizacijo, ki bo spodbujala ponovno uporabo kode. Namesto da bi nek modul posamezni programer napisal sam, se mu mora bolj izplačati, da poišče in modificira primerno že zgrajeno komponento. Ponovna uporaba programske opreme zahteva dodatne investicije, ki se poplačajo šele v nekaj letih. Razvoj ponovno uporabnih komponent namreč zahteva več časa in dela kot razvoj komponent za enkratno uporabo. Knjižnica ali podatkovna zbirka ponovno uporabnih komponent se lahko zgradi le v daljšem času. Gradnja take knjižnice se izplača, če bo razvojna skupina v prihodnosti izdelala več podobnih sistemov. Uvajanje raznih orodij (generatorjev kode, aplikacijskih okvirjev itd.) pa zahteva veliko dodatnega učenja,

ki se bo izplačala šele po izdelavi več sistemov z novo tehnologijo.

12.3 Umetna inteligenca

Velik vpliv na obliko in način razvoja programske opreme bodo v prihodnosti igrale metode, ki jih razvija *umetna inteligenca* [53]. Za uporabniške vmesnike bo pomembno računalniško razumevanje naravnega jezika in prepoznavanje govora ter avtomatična interpretacija slik. Raziskovalci na področju umetne inteligence se ukvarjajo tudi z dokazovanjem pravilnosti programov, razvojem jezikov na zelo visokem nivoju (angl. Very High Level Languages), specifikacijami v naravnem jeziku, programiranjem na osnovi primerov in z inteligentnimi asistenti. Inteligentni asistenti so ekspertni sistemi za pomoč pri programiranju. V ekspertnih sistemih je zajeto strokovno znanje določenega področja. Ekspertni sistemi za pomoč pri programiranju pomagajo pri načrtovanju in razvijanju programske opreme. Elemente ekspertnih sistemov vsebujejo številna sodobna orodja CASE.

12.4 Multimediji in vizualizacija

Na obliko programske opreme, predvsem na obliko vmesnika med človekom in strojem, vplivajo ideje *vizualizacije*. Vizualizacijo vodi spoznanje, da s slikami lahko posredujemo zelo veliko informacij v zelo kratkem času, saj človek interpretira slike celostno, ne pa linearno kot tekst in številke. V poplavi podatkov, ki so rezultat raznovrstnih simulacij, lahko uporabnik kvantitativno preveri le majhen del rezultatov — kar pomeni, da je nemogoče celostno hitro analizirati številčne rezultate. Če pa številke prevedemo v obliko slike ali celo zaporedja slik, lahko opazujemo na primer pojav turbulenc v hidrodinamiki, preučujemo rezultate novih medicinskih diagnostičnih metod ipd. Vizualizacija ni pomembna le na tradicionalnih področjih, kjer že sicer uporabljamo slike in grafično predstavitev. Danes poskušamo vizualizirati in animirati procese in algoritme, ki nimajo naravne upodobitve. Vizualizacija vpliva ne le na oblikovanje uporabniških vmesnikov in na način predstavitve rezultatov posameznih uporabniških programov, temveč omogoča tudi nov način vizualnega programiranja. Vizualizacija je le del *multimedijev*, to je interaktivnega združevanja in prepletanja teksta, grafike, animacije, zvoka in videa, kar omogoča povsem novo kvaliteto pri uporabi računalnikov za učenje, izražanje in pridobivanje izkušenj [5, 61]. Zmožnost vizualizacije in večje procesorske moči ima tehnika navidezne resničnosti (angl. virtual-reality), ki s pomočjo programske opreme in poseb-

nih vmesnikov med računalnikom in človeškim telesom (vid, tip, zvok in gibanje delov telesa) simulira resničen ali povsem izmišljen svet.

12.5 Medmrežje

Globalno računalniško omrežje ali medmrežje (angl. Internet), predvsem pa svetovni splet (angl. World Wide Web), beležita v zadnjih nekaj letih skokovit razvoj. Svetovni splet je populariziral medmrežje in ga spremenil iz predvsem znanstvenega foruma v informacijski, ekonomski in umetniški forum [70]. Svetovni splet je s svojo multimedijsko naravnanostjo, širokim spektrom uporabe, vizualno atraktivnostjo in enostavnim načinom dostopa do informacij postal vseprisoten način komuniciranja in izmenjevanja informacij. Svetovni splet postaja povsem nov komunikacijski medij, ki svojo pravo podobo še oblikuje [60]. Skokovit razvoj medmrežja je presenetil skoraj vse razvijalce zasebnih ali internih informacijskih sistemov, ki so temeljili na različnih programskih in celo strojnih platformah. Danes se tako rekoč vsi prilagajajo tehnologiji medmrežja, kar interne informacijske sisteme naredi neodvisne od strojne in programske opreme. Interni informacijski sistemi, ki so od javno dostopnega medmrežja ločeni z ustreznimi varnostnimi mehanizmi, so dobili ime intranet.

Z vidika razvoja programske opreme je medmrežje prineslo veliko sprememb. Novi programski jezik java omogoča skoraj popolno prenosljivost programske opreme. Pojavile so se potrebe po povsem novih vrstah programske opreme. Zaščita podatkov je postala kritični dejavnik. S pomočjo medmrežja je koordinacija razvoja programske opreme lažja. Razvojna skupina je lahko razkropljena po vsem svetu. Uporabniki programske opreme lahko enostavno in hitro dobijo pomoč razvijalcev ali skupine za podporo. Medmrežje omogoča lažji in mnogo cenejši način distribucije programske opreme. Možni so novi načini uporabe programske opreme, ko programske opreme ni potrebno kupiti, plačujemo lahko le njeno vsakokratno uporabo. Povsem nova vrsta programske opreme so inteligentni agenti [55]. Agenti delujejo kot naši pomočniki, ki po naših navodilih samostojno iščejo informacije po medmrežju ali nam pomagajo filtrirati informacije, s katerimi nas zasipava medmrežje. Skratka, dela za programerje še ne bo zmanjkalo!

12.6 Etična vprašanja

Računalniki človeški družbi žal niso prinesli le koristi. Uporaba računalnikov povzroča številne probleme in odpira moralne dileme, ki se jih morajo zave-

dati tudi razvijalci in uporabniki programske opreme [15, 16]. Celotna človeška družba je zaradi odvisnosti od računalniških sistemov vedno bolj ranljiva. Hude težave lahko povzročijo napake v programu ali fizični napadi na računalniške sisteme. Med novimi družbenimi problemi so najbolj pereči: računalniški kriminal in varnost računalniških sistemov; kraja programske opreme in varstvo intelektualne lastnine; računalniški virusi; nezanesljivost programske opreme in škoda, ki jo lahko povzroči napaka v programski opremi; podatkovne zbirke in varstvo zasebnosti; družbene posledice ekspertnih sistemov; številni problemi v zvezi z informatizacijo delovnih mest.

Literatura

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, Reading, MA, 1982.

[2] Air Force Studies Board. *Adapting Software Development Policies to Modern Technology*, Washington, D.C., 1989. National Academy Press.

[3] M. Alford. SREM at the age of eight: the distributed computing design system. *Computer*, 18(4):36–46, 1985.

[4] *Annals of the History of Computing*, volume 1(6), 1984.

[5] E. Barrett, editor. *Text, ConText, and HyperText*. MIT Press, Cambridge, MA, 1988.

[6] B. W. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.

[7] B. W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72, 1988.

[8] G. Booch. *Object-Oriented Analysis and Design with Applications, Second Edition*. Benjamin/Cummings, Redwood City, CA, 1994.

[9] F. P. Brooks. *The Mythical Man-Month*. Addison-Wesley, Reading, MA, 1975.

[10] T. DeMarco. *Controlling Software Projects*. Yourdon Press, 1982.

[11] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[12] E. Downs, P. Clare, and I. Coe. *SSADM: Structured Systems Analysis and Design Method*. Prentice-Hall, 1988.

[13] J. C. Emery. *Management Information Systems*. Oxford University Press, New York, 1987.

[14] R. Fischer and W. Ury. *Getting to YES*. Penguin, New York, 1983.

[15] T. Forester, editor. *Computers in the Human Context*. MIT Press, Cambridge, MA, 1989.

[16] T. Forester and P Morrison. *Computer Ethics, Cautionary Tales and Ethical Dilemmas in Computing, Second Edition*. MIT Press, Cambridge, MA, 1994.

[17] D. A. Garvin. What does "product quality" really mean? *Sloan Management Review*, (Fall), 1984.

[18] T. Gilb. *Principles of Software Engineering Management*. Addison-Wesley, Wokingham, England, 1988.

[19] E. T. Hall. *The Hidden Dimension*. Doubleday, Garden City, NY, 1969.

[20] A. Hauc. *Projekti v organizacijah združenega dela*. ČGP Delo, TOZD Gospodarski vestnik, Ljubljana, 1982.

[21] A. Hauc, editor. *Projekti in razvoj, I. posvetovanje v Ljubljani*, Maribor, 1987. Visoka ekonomska šola.

[22] N. Havelka. *Psihološke osnove grupnog rada*. Naučna knjiga, Beograd, 1980.

[23] F. J. Heemstra. *How Much Does Software Cost*. PhD thesis, Kluwer, 1989.

[24] R. Hirscheim and H. K. Klein. Four paradigms of information systems development. *Communications ACM*, 32(10):1199–1216, 1989.

[25] W. S. Humphrey. Characterizing the software process: a maturity framework. *IEEE Software*, 5(2):73–79, 1988.

[26] IEEE guide to software requirements specifications, IEEE std 830, 1984.

[27] ISO standards 9000–9004: ISO 9000: Quality management and quality assurance standards—guidelines for selection and use. ISO 9001: Quality systems—model for quality assurance in design/development, production, installation and servicing. ISO 9002: Quality systems—model for quality assurance in production and isntallation. ISO 9003: Quality systems—model for quality assurance in final inspection and test. ISO 9004: Quality management and quality system elements—guidelines. ISO, 1987.

[28] M. A. Jackson. *System Development*. Prentice-Hall, 1983.

[29] R. E. Johnson. Frameworks = (components + patterns). *Communications ACM*, 40(10):39–42, 1997.

[30] D. E. Knuth. *The TEXbook*. Addison-Wesley, 1994.

[31] J. Knutson and L. Glauber. *Using SuperProject Plus*. Osborne McGraw-Hill, Berkeley, 1987.

[32] I. Kononenko. *Načrt podatkovnih struktur in algoritmov*. Založba FE in FRI, Ljubljana, 1996.

[33] I. Kononenko. *Logično programiranje in drugi principi programskih jezikov*. Založba FE in FRI, Ljubljana, 1997.

[34] D. W. Lang. *Critical Path Analysis*. Hodder and Stoughton, London, 1977.

[35] M. M. Lehman and L. A. Belady, editors. *Program Evolution*. APIC Studies in Data Processing No. 27. Academic Press, 1985.

[36] H. A. Levine. *Project Management Using Microcomputers*. McGraw-Hill, Berkeley, CA, 1986.

[37] S. Levy. *Hackers, Heroes of the Computer Revolution*. Anchor Press/Doubleday, Garden City, NY, 1984.

[38] D. A. Marca and C. L. McGowan. *SADT, Structured Analysis and Design Technique*. McGraw-Hill, 1988.

[39] A. Maslov. *Motivacija i ličnost*. Nolit, Beograd, 1982.

[40] J. A. McCall, P. K. Richards, and G. F. Walters. Factors in software quality. Technical Report RADC-TR-77-369, US Department of Commerce, 1977.

[41] G. A. Miller. The magical number seven, plus or minus two: some limits on our capacity for processing information. *Psychological Review*, 63:81–79, 1956.

[42] H. Mintzberg. *Structures in Fives: Designing Effective Organizations*. Prentice-Hall, 1983.

[43] P. Naur and B. Randell, editors. *Software Engineering*, Garmisch, 1968. NATO Scientific Affairs Division.

[44] E. A. Nelson. Management handbook for the estimation of computer programming costs. Technical Report AD-A648750, Systems Development Corporation, 1966.

[45] D. A. Norman. Some observations on mental models. In D. Gentner and A. L. Stevens, editors, *Mental Models*, pages 7–14. Erlbaum, 1983.

[46] K. T. Orr. *Structured Requirements Definition*. Ken Orr & Associates, Inc., Topeka, KS, 1981.

[47] M. Page-Jones. *The Practical Guide to Structured Systems Design*. Yourdon Press, 1988.

[48] D. L. Parnas. On criteria to be used in decomposing systems into modules. *Communications ACM*, 14(1):221–227, 1972.

[49] D. L. Parnas. Designing software for ease of extension and contraction. In *Proceedings 3rd International Conference on Software Engineering*, pages 264–277. IEEE, 1978.

[50] C. Pescio. Principles versus patterns. *Computer*, 30(9):130–131, 1997.

[51] R. S. Pressman. *Software Engineering*. McGraw-Hill, New York, 1987.

[52] W. J. Reddin. *Managerial Effectiveness*. McGraw-Hill, 1970.

[53] C. Rich and R. C. Waters, editors. *Readings in Artificial Intelligence and Software Engineering*. Morgan Kaufmann, Los Altos, 1986.

[54] Y. Rogers, H. Sharp, D. Benyon, S. Holland, and T. Carey. *The Human–Computer Interaction*. Addison-Wesley, Wokingham, 1994.

[55] J. S. Rosenschein. *Rules of Encounter*. MIT Press, Cambridge, MA, 1994.

[56] N. Rot. *Psihologija grupa*. Zavod za učbenike i nastava sredstva, Beograd, 1983.

[57] B. Shneiderman. *Designing the User Interface*. Addisson-Wesley, Reading, MA, 1992.

[58] G. E. Sievert and T. A. Mizell. Specification-based software engineering with TAGS. *Computer*, 18(4):56–65, 1985.

[59] C. J. Sindermann. *Winning the Games Scientists Play.* Plenum, New York, 1982.

[60] M. Stefik. *Archetypes, Myths, and Metaphors.* MIT Press, Cambridge, MA, 1996.

[61] R. Steinmetz and K. Nahrstedt. *Multimedia: Computing, Communications and Applications.* Prentice-Hall, 1995.

[62] W. Stevens, G. Myers, and L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, 1974.

[63] E. B. Swanson and C. M. Beath. Departmentalization in software development and maintenance. *Communications ACM*, 33(6):658–667, 1990.

[64] D. Teichroew and E. Hershey. PSL/PSA: A computer aided technique for structured documentation and analsis of information processing systems. *IEEE Transactions on Software Engineering*, 3(1):41–48, 1977.

[65] H. van Vliet. *Software Engineering Principles and Practice.* Wiley, Chichester, 1993.

[66] I. Vessey. Expertise in debugging computer programs: a process analysis. *International Journal of Man-Machine Studies*, 23:459–494, 1985.

[67] A. Vila. *Planiranje proizvodnje i kontrola rokova.* Informator, Zagreb, 1972.

[68] B. Šali. *Osnove psihologije.* Dopisna delavska univerza, Ljubljana, 1973.

[69] J.-D. Warnier. *Logical Construction of Programs.* Stenferd Kroese, 1974.

[70] A. Weintraub. Art on the web, the web as art. *Communications ACM*, 40(10):97–102, 1997.

[71] N. Whitten. *Managing Software Development Projects.* Wiley &Sons, New York, 1990.

[72] N. Wirth. Program development by stepwise refinement. *Communications ACM*, 14(4):221–227, 1971.

[73] S. J. Young. *Real Time Languages: Design and Development*. Ellis Horwood, Chichester, 1982.

[74] E. Yourdon and L. Constantine. *Structured Design*. Yourdon Press, 1978.

Knjigo je uredil in oblikoval Franc Solina
s sistemom LaTeX